USENIX LARGE INSTALLATION SYSTEMS ADMINISTRATION III WORKSHOP AUTUMN 1989

U

US

USE

U NIX

USENIX

# WORKSHOP PROCEEDINGS

## LARGE INSTALLATION SYSTEMS ADMINISTRATION III

September 7-8, 1989
Austin, TX

For additional copies of these proceedings write

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $13

Outside the U.S.A and Canada, please add
$9 per copy for postage (via air printed matter).

# Program and Table of Contents
# Workshop on Large Installation Systems Administration III
# September 7-8, 1989

## Thursday, September 7

## Friday, September 8

---

# Site

## A Language and System for Configuring Many Computers as One Computing Site

Bent Hagemark
Kenneth Zadeck
Department of Computer Science
Brown University
Providence, RI 02912

**Abstract**

This work describes the usage, design and implementation of a language and system for managing the configuration of many computers which together form a single computing site.

## 1. Introduction

Many modern computing sites consist of a large number of computers networked together. For the administrator of such a computing site this introduces many problems not encountered in the previous generation of large central time sharing systems. Among the issues faced in ensuring that a site of many networked computers works as intended is the problem of maintaining an accurate and consistent configuration of all computers in the site. There are few solutions which elegantly address the broad scope of specifying configuration information on a site-wide basis for a site of many computers which must cooperate with each other and with common resources.

This work proposes a solution to the specification problem involved in configuring the subsystems found in a typical site consisting of networked UNIX workstations and servers. A simple utility uses this specification to produce the configuration files for all the computers in the site. This tool is intended for use by computer novices as well as expert systems programmers. The solution proposed here could also be used as the foundation of "plug and play" tools which would permit highly automated system startup procedures for teaching a new computer (and user!) about the available services at that site.

### 1.1. Terminology

Before proceeding we will clarify some terms.

| | |
|---|---|
| **computer** | An individual workstation, server or traditional computer. A part of a site. |

| | |
|---|---|
| **site** | One or more computers and peripherals all under the same ownership or administrative domain. The computers and peripherals of a site are often interconnected with one or more networks. |
| **network** | The physical local area or long distance methods for connecting computers to each other. Not all computers on a particular network necessarily belong to the same site. |
| **installation** | This term refers to the activity of setting up a computer or site. |

## 1.2. Overview

After briefly introducing the problem we detail the Sitefile format (or "mini language") and how this may be used as a solution to the problem. We describe a simple utility based on this language and suggest future areas of development.

## 2. The Problem

Managing the information used to produce configuration files is the central problem in maintaining the configuration files of all the computers in a site consisting of networked UNIX workstations and servers. Files such as /etc/fstab, /etc/hosts--including YP maps or BIND database files--/etc/printcap, kernel config files together with the files and directories to which some of these files refer repeat many individual pieces of information. Information such as hostnames, network addresses, printer names, and file system directories must be kept consistent between different configuration files on the same computer as well as potentially on all computers in the site. Updating these files to reflect changes in the site is consequently tedious and prone to error. In addition to these semantic errors are the more basic errors at the syntactic level. Many configuration files demand the same sensitivity to syntax as typical programming languages and thus require programmer-level expertise to manipulate them.

Reducing the chances of error is very important as inconsistency of information or improper syntax in configuration files can lead to systems or network failure.

## 3. The Solution

The solution presented here takes a step back from the configuration files themselves and focuses on managing the information inherent in configuration files-- the goal being to automate the production and maintenance of configuration files. The key design feature in automating this process is to generate all configuration files from a common representation.

The common representation is described in a Sitefile. The site utility takes this as input and outputs all configuration files for all the computers in the site. The program site is a "configuration file compiler" of sorts with the format of the Sitefile being the "language". (At the end of this paper we propose other tools for use with Sitefiles). See Figure 1 for an illustration of the framework of the solution.

## 3.1. Introduction to Sitefiles

Before discussing how a Sitefile is used in site administration we first briefly introduce the features of the language describing the format of a Sitefile. A Sitefile consists *variable* and *type* entries. A variable or type has an

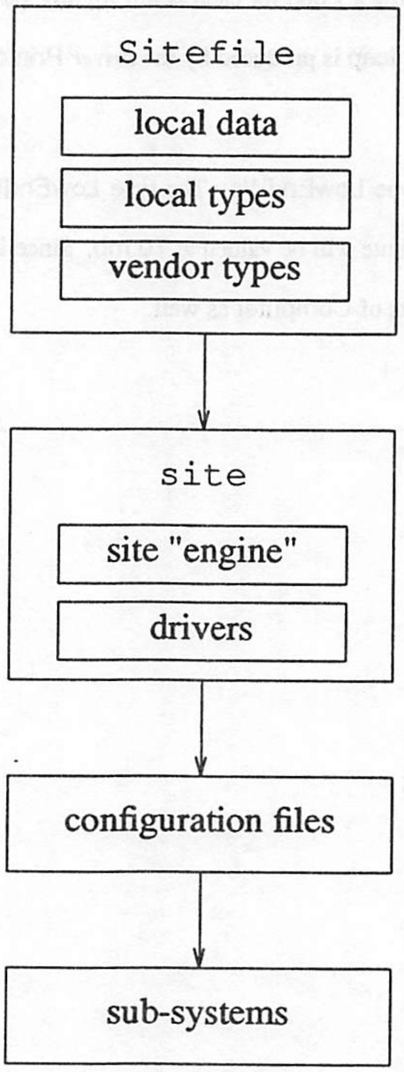

*Figure 1: Framework of the solution*

associated body which consists of a set of *attributes* and *components*. A variable is defined to be of a particular type. A type declaration associates a name with set of attributes and components. A type declaration may in turn be based on other types which imply the standard semantics of multiple inheritance for the attributes and components.

The first entry in Figure 2 is a variable. It defines the variable Bob to be of the type Computer. Unique to Bob are the attributes cpu and memsize valued at risc and 16 mb respectively. The keyword class introduces the second entry in Figure 2. This entry declares the type Computer to have 4 attributes (cpu, memsize, display, disk) and one component (/etc/printcap). Any variable based on Computer will take on the values specified for the attributes display and disk and will inherit the component /etc/printcap. Leaving the value of the cpu and memsize attributes blank implies that setting a value for each is left for any variable of type Computer or type based on Computer. The component /etc/printcap is produced by the *driver* Printcap. (We will cover components in more detail later).

The variable Mary is of the type LowEndWs. The type LowEndWs is based on (or *derived* from) Computer and specifies that the disk attribute will be valued at 70 mb. Since LowEndWs is based on Computer Mary takes on the attributes and components of Computer as well.

```
Computer Bob {
        cpu =           risc;
        memsize =       16 mb;
}

class Computer {
        cpu =           ;
        memsize =       ;
        display =       color;
        disk =          140 mb;
        /etc/printcap := Printcap();
}

LowEndWs Mary {
        cpu =           cisc;
}

class LowEndWs : Computer {
        disk =          70 mb;
}
```

*Figure 2: Example Sitefile entries*

In Figure 3 we give the general form of a variable and a type.

This should give some feeling for what the language looks like. The language is declarative and thus includes no facility for specifying action (sequences of instructions), conditionals or loops. There are no built in variables, types, attributes or components. Attributes and components have no type and the names of attributes and components are independent of the names of types and variables.

## 3.2. Use of Sitefiles

A Sitefile describing a computing site is typically organized in 3 levels. Fundamental to the use of Sitefiles for this purpose is the notion that all objects in a computing site can be classified and that all information about the site and the objects in it can be represented by simple attribute-value associations. As illustrated in Figure 1, the information can be organized into 1) local data, which directly reflect the "first class" objects which vary from site to site; 2) local types, which describe the policies and customization particular to a single site; and 3) vendor types, which describe the default configurations of a vendor's product line. Each level is typically organized into its own file. The separate files are #include'ed into a Sitefile for use as input to site.

This layered approach is an important part of the design. The intent is to separate the parts of the representation requiring no programming expertise from those parts which require the expression of conditionals, loops and sequences. Working at the highest level requires the least amount of expertise. It is also the level at which one sees the most activity in the day to day use of site. This highest level is targeted at non-programmer systems administrators. The middle and lowest layers of the Sitefile are more complex and constitute the "programming" of an individual site and of a vendor's product offerings respectively. The entries at the lowest level of a Sitefile remain constant across all sites using products from that vendor.

```
/* general form of a variable */
<type-name> <variable-name> {
        <body>
}

/* general form of a type */
class <type-name> [ : <super-type-names> ] {
        <body>
}
```

*Figure 3: Example Sitefile entries*

In the next few sections we examine the use of the "mini-language" which constitutes the format of a Sitefile. We will highlight the language features as we present Sitefile programming in three levels. We will work from the top down and afterwards describe how a Sitefile is processed by the site utility.

## 3.3. Sitefile: Top Level -- Variables

Sitefile entries at this highest level are the data directly reflecting the inventory of computers, printers, networks, and other such "first class" objects constituting the computing site. That is, these items are what vary from site to site and are represented by Sitefile *variables*. Each defined variable corresponds to exactly one "first class" part of the site. Manipulations of Sitefile variables generally reflect the common tasks of adding and removing objects from the site. For example, installing a new computer in the site implies the simple addition of a variable definition to this part of the Sitefile. As seen in Figure 4 this entry in the Sitefile defines the computer john to be of the type Ws and specifies the values unique to it. In this example the attribute hostname has the value john.

## 3.4. Sitefile: Middle Level -- Local Types

Sitefile entries at this level represent the "programming" of customization and tailoring peculiar to an individual site. In Sitefile syntax these entries take the form of types. The variables at the top level are defined in terms of the types declared at this level. And, the types at this level are in turn derived from types found at the lowest (vendor) level.

See Figure 5 for an example Sitefile entry at this level. This entry declares a site-specific classification (type) of a computer based on a classification declared by the manufacturer.

The empty value fields in the attributes named hostname, netnumber, and loc indicate that variables of the type Ws must specify their own values for these "virtual" attributes. Any variable based on this type also takes on

```
Ws john {
        hostname =      john;
        netnumber =     128.148.35.42;
        loc =           CIT 576;
}
```

*Figure 4: Example Sitefile local data entry*

the specified values of the cpu, display, and disk attributes.

The final field in the type declaration in Figure 5 specifies that any variable based on the type Ws will use the EtcHosts driver to produce the contents of its /etc/hosts file. This introduces the use of Sitefile *components* to represent configuration files. The name to the left of the ":=" corresponds to the configuration file name. The syntactic form on the right hand side of the ":=" identifies the driver used to produce the contents of the file specified on the left.

Information at this level also describes site-wide policies and services. For example, one could specify whether to use /etc/hosts, YP or BIND as the hostname lookup service by classifying a computer as a YP client or YP server, a BIND client or server, or as a client or server of neither service. See Figure 6 for a example of type declarations illustrating this technique. Note that the type Ws has now been further classified as a YPClient in addition to being a VendorWs. In object oriented terminology this is *multiple inheritance*.

A variable of a type derived from YPClient will receive an /etc/hosts produced by the StubEtcHosts() driver. (During normal operation a YP client uses the YP hosts map for hostname to network number mapping; however, for the boot process to work properly the client must have a "stub" /etc/hosts defining at least its own hostname to IP number mapping.) Variables based on YPServ will receive an /etc/hosts file produced by FullEtcHosts().

## 3.5. Sitefile: Low Level -- Vendor Types

Sitefile entries at this lowest level classify a vendor's product offerings. See Figure 7 for an example type declaration at this level. The cpu, display, and disk are virtual attributes implying that sub-types or variables of

```
class Ws : VendorWs {
        hostname =        ;
        netnumber =       ;
        loc =             ;
        cpu =             68020;
        display =         color 8 planes;
        disk =            scsi 100 mb;
        "/etc/hosts" :=   EtcHosts();
}
```

*Figure 5: Example Sitefile local type entry*

```
class Ws : VendorWs YPClient {
        ...
}
class Serv : VendorServ YPServ {
        ...
}
class YPClient {
        ...
        "/etc/hosts" :=   StubEtcHosts();
}
class YPServ {
        ...
        "/etc/hosts" := FullEtcHosts();
}
```

*Figure 6: Example Sitefile local policy entry*

VendorWs must supply the values for these attributes. All variables based on VendorWs will inherit the netdev ı to the value of ethernet. The ethernet value to the netdev attribute specifies the networking technology used on that vendor's workstations.

## 3.6. Site program: Configuration File Drivers

We now turn to a brief discussion of the drivers used to produce the contents of configuration files. Drivers are part of the implementation of the Site program. This section corresponds to the "drivers" box within "site" in

```
class VendorWs {
        cpu =              ;
        display =          ;
        disk =             ;
        netdev =           ethernet;
        "/etc/fstab" :=    VendorFStab();
        "/vmunix" :=       VendorKernel();
        "/etc/ttytab" :=   Ttytab();
}
```

*Figure 7: Example Sitefile vendor type entry*

```
EtcHosts(target, arglist, thisvar, ofile)
char *target;
LISTP arglist;
char *thisvar;
char *ofile;
{
        FIO fio;
        LIST vl;
        char *varn;
        ITER iter;

        LIST hostname;                  /*
        LIST hostnumber;                 * attributes
        LIST ipsubnet;                   */

        SITEfioopen(&fio, ofile);
        SITEfioprintf(&fio, "# /etc/hosts for %s\n", thisvar);
        SITEfioprintf(&fio, "# generated from %s\n", ofile);

        SITEvarnamelist(&vl, thisvar);

        SITElistiterinit(&vl, &iter);
        while ( varn=(char*)SITElistiter(&vl, &iter) ) {
                SITEattrval(&hostname, varn, "hostname");
                SITEattrval(&hostnumber, varn, "hostnumber");
                SITEattrval(&ipsubnet, varn, "ipsubnet");

                if ( SITElistlen(&hostname) == 0 )
                        continue;

                SITEfioprintf(&fio, "%s\t%s.%s\n",
                        SITElistfirst(&hostname),
                        SITElistfirst(&ipsubnet),
                        SITElistfirst(&hostnumber));
        }
        SITEfioclose(&fio);
}
```

*Figure 8: Example driver*

Figure 1.

A driver is a C function which uses a set of Sitefile access routines to look up attribute values in variables as the source of information needed to produce the corresponding configuration file. For example, the VendorFStab driver looks for disk attributes of the variable for information needed to produce an /etc/fstab file. Additionally, the VendorKernel accesses the same information for its needs in producing a /vmunix file. See Figure 8 for an example driver for a full /etc/hosts file.

The Hosts driver in Figure 8 produces a file (named in ofile) in the format of an /etc/hosts file. SITEvar-namelist() returns a list of names of all variables defined in the Sitefile. The while loop iterates through this list looking up the values of the hostname, hostnumber, and ipsubnet attributes for each variable. Any variable with a hostname attribute will yield an entry in the file.

## 3.7. Sitefile: Summary of Features

Several important points central to the design of site and Sitefiles are illustrated in the preceding sections. We summarize these points here.

### 3.7.1. High Level

This tool is designed to be used by people who are **not** computer experts. This tool permits manipulation of high level data to effect low level changes. The data manipulated at the each level of interaction is consistent with the expertise needed to manipulate it. The principle underlying this design is that there there is a set of defaults which will result in a properly functioning system of that type and in a properly functioning site overall.

### 3.7.2. Factoring of Information

There are two features in the design to solve the problem of redundant information. First, the use of type hierarchies to factor common information makes it possible to specify a particular piece of information at any level of classification. Second, the Sitefile access routines used in the drivers allow different configuration files to retrieve a common piece of information. For example, an individual computer's hostname is often found in many configuration files on that computer as well as in the configuration files found on potentially all of the computers in the site. Each configuration file driver knows to find a computer's hostname by looking up the "hostname" attribute in the variable associated with that computer.

## 3.8. Use of the Site Program

The site program processes a Sitefile be "evaluating" each variable in the file. Evaluating a variable further "evaluates" each component of the variable. Evaluating a component is carried out by calling the *driver* associated with that component.

In client-mode site connects to a "Sitefile server" and evaluates the variable associated with the local host thus producing the configuration files for only that computer. Otherwise site evaluates all variables in the Sitefile producing all configuration files for all computers along with a Distfile for distributing them.

The client-mode implementation provides the foundation for a "pull" style update mechanism. This is important in a very large site where "pushing" files out from a single machine is infeasible. Client-mode site called at boot time could be used to completely automate the installation and local customization of machines new to the site.

## 4. Other Approaches

This design stems largely from the experience of trying to use rdist and Distfiles to manage the configuration of a Computer Science Department site consisting of 100+ workstations and servers from several vendors. The strategy here was to gather up all configuration files for all machines into a directory on a server. A very complex Distfile (~1000 lines) described how these files where to be installed on various machines. The most difficult problem in using rdist for this task was in trying to impose a classification scheme for the various types of machines in the site. Beyond the basic mechanism of providing the actual transport of files to remote machines there are few facilities for actually producing the configuration files let alone the managing of the information needed to produce the files.

Interactive user-friendly front-ends often simplify the initial installation of a computer. The problem with this approach is that these utilities place any information gathered from the administrator directly into configuration files. This severely limits extensibility and local customization possibilities. In following the Sitefile approach any such interactive front-ends would instead operate on a Sitefile allowing the utility to focus on correctness of the information and at the same time relieve burden of properly formatting a configuration file.

## 5. State of Implementation

The site utility is a prototype implementation at this time. Some drivers have been implemented to demonstrate the concepts discussed in this paper.

## 6. Future Work

The current implementation of site requires one to relinquish control of configuration files to site and a Sitefile. A facility for "reverse compiling" existing configuration files coupled with some heuristics for resolving conflicts between the information inherent in a configuration file and the information in a Sitefile would permit direct editing of configuration files under the control of site. A more important use for such a capability would allow implementation of a finer grain "incremental" change mechanism. That is, site would be smart enough to generate only those files which need changing based on an incremental change to the Sitefile.

We do not address network security or authentication. The Sitefile "language" described in this work does not allow expression of the concepts of administrative domain and ownership of parts of a site. One could possibly extend the language to permit ownership of update permission to types, variables, components or attributes and extend the implementation of site to use this information-- presumably along with a network authentication mechanism.

The most radical implementation would call for the outright removal of all configuration files. The functionality of attribute lookup found currently in the drivers could instead be moved to the C library functions used to peruse the "/etc/blah" file. For example, the implementation of the getfsent() routine could be replaced by (network transparent) Sitefile calls. This already has precedence in the implementation of C library functions such as the getpw*() and gethost*() for use with YP.

The current implementation of site uses Sitefile components to represent configuration files. This implementation could be easily extended to permit use of Sitefile components to describe specific directories which need to exist with certain mode, owner and group settings, as well as special--/dev--files, and finally for actual software "subsets". Rdist Distfiles already can handle the transport issues of most of these situations. A more comprehensive implementation of site would allow processing of Sitefile entries such as the one in Figure 9.

## 7. Conclusion

This paper has described a solution to one of the problems faced in managing a site consisting of many computers. We have identified the problems in managing configuration files on a site-wide basis and have proposed a solution to managing the information inherent in these configuration files which automates the managing of these files on a site-wide basis. We hope that tools such as site will see wide-spread use in simplifying the task of managing sites consisting of many computers.

```
class Ws {
        hostname =        ;
        ...

        "/usr" :=         Usr();
        "/dev" :=         Dev();
        "/usr/local" :=   UsrLocal();
        ...

        MountPoints();  /* fstab dirs */
        LpdSpoolDirs(); /* printcap dirs */
}
```

*Figure 9: Future Sitefile entry*

## 8. References

[1] Sun Microsystems, *System and Administration Guide, Chapter 14: The Sun Yellow Pages Service,* SunOS 4.0, May 1988.

[2] UC Berkeley, BSD 4.3 Unix User's Manual Reference Guide (URM), *rdist(1)*, November 1986.

# Tools for System Administration in a Heterogeneous Environment

Raphael Finkel Brian Sturgill

**Abstract**

System administration in large sites must maintain a coherent organization of software across many machines of different architectures and operating systems. This paper describes the **SAT** package of tools intended for system administrators. These tools are centered around a distributed database manager that can store data needed in administration, such as that pertaining to hosts and users. The database provides replication, access control, and locking. It can be queried directly by programs in the **SAT** package and by programs written in **SAL**, a simple command and query language. The paper presents non-trivial examples of **SAL** programs for building configuration files, forcing file consistency, distributing software, and monitoring resource usage. It describes how database relations are organized, replicated, locked, and protected. This package is currently in use at the University of Kentucky.

## 1 Introduction

This paper describes a set of system administration tools (**SAT** and **SAL**) developed from our experience in running several varieties of Unix. These tools allow a small staff to maintain a very large number of machines, even in a heterogeneous environment that covers several flavors of Unix. The tools are general enough to be used in environments with other operating systems, but we have not tested such extensions.

**SAT** is a special-purpose database package that allows us to maintain centralized data describing the individual hosts, the user community, the available operating system varieties, and the peripherals. A query and command language, **SAL**, converts these data to the form required by each operating system. This language has facilities to perform software maintenance tasks as well as to generate command procedures (in other languages) customized for differences between machines.

We begin by defining the system administration problem. We then introduce our solution. We show how this solution addresses the individual components of the problem by presenting examples drawn from actual practice. We discuss implementation techniques and comment on how well our tools will work in more disparate environments. We close with a review of related efforts and a status report.

## 2 The Problem

This project attempts to organize the administration task in a large, heterogeneous installation. We will call the collection of all computers the **site**. The individual machines of the site are called **hosts**. Hosts are classified into **families** based on hardware type and operating system type. For example, our site contains four families: Symmetry Sequent running Dynix, Sun 3 running SunOS, DEC Vax running Ultrix, and AT&T 3Bx running System V. The hosts within a family differ primarily with respect to attached peripherals, such as monitor type, disk size, and presence of printers and tape drives.

The task of system administration (we will just say **administration**) is to maintain a coherent organization of software across the site. This task has many subparts:

- Building configuration files. Many operating systems, Unix in particular, run programs that expect to find data describing the environment in well known **configuration files**. For example, machines directly accessible through networks are listed in `/etc/hosts`, printers in `/etc/printcap`, the organization of the disk in `/etc/fstab`, and the characteristics of users in `/etc/passwd`.

  The content of these files tends to be similar but not identical across a site. For example, a host directly connected to a shared printer needs specific information such as the communications port to which the printer is attached. Other hosts need to know only the name of the printer's machine and how to reach it over the network.

  Configuration files must adhere to consistency constraints. For example, each host in the configuration file that describes trusted hosts (`/etc/hosts.equiv`) must be in `/etc/hosts`.

- Keeping the file system coherent. The first problem is checking for consistency within rules specified by the administrator. For example, an integrated environment suggests that each host remotely mount directories in a manner consistent with other hosts, so a user may log into any host and perceive the same file structure. Nonetheless, the actual executables that are mounted should match the host's family. Another example is that each user must own its home directory. For security reasons, every user should own a startup file protected from modification by others, even if this file is empty.

  A second problem has to do with disk consistency. Missing files must be tracked down, and corrupted files must be fixed. Configuration files in particular must be kept accurate and uncorrupted.

- Distributing changes and new software. Bringing a new machine up often involves supplementing a vendor-supplied software set with site-specific configuration information and software. The support programs supplied by vendors for this activity usually assume that each host is self-sufficient, or at best, that every host has the same special programs and procedures. But a large computing facility might contain hundreds of different machines. The user, hardware, and application environments of such a facility are rarely static. Changes need to be propagated to each host on a regular basis, often in a different data storage format for each vendor.

  Software upgrades sometimes apply only to a given family, but much software is intended to run on all families. For example, the text editor might be identical across all families. When software is introduced or changed, it must be (re)compiled and tested for all relevant families and then distributed. The distribution may touch hundreds of machines; each must get its correct version. Other software that depends on newly modified software must also be upgraded.

  It is often necessary to reboot a certain subset of the hosts at the same time. In a similar vein, software upgrades and fixes within a family often require running a program across all hosts in that family.

- Monitoring resource usage and balancing load. Resources include cpu cycles, disk space, specialized peripherals, virtual memory, and the right to execute licensed programs. Each of these resources may be monitored for usage patterns on each host. Balancing load involves creating rules to allocate limited resources. For example, the allocation of file systems to file servers should be balanced to prevent communication and I/O bottlenecks. As another example, a rule might dictate that when a particular application uses more than a few cpu seconds, it should be given high priority.

Our definition of administration only covers a small part of the actual responsibility of real administrators. They must manage staff, respond to user requests, choose hardware, keep licenses and maintenance contracts current, organize the logical structure of hardware and software, and remain aware of new software that becomes available. Our tools do not address these responsibilities, although it is certainly conceivable that the database tools we will discuss could record such information as staff jobs, license agreements, maintenance contracts, and logical structure.

## 3 The Solution

**System Administration Tool** (**SAT**) is a distributed database manager that stores descriptive data needed in administration. The database is composed of **relations**, each of which is described by a **scheme** and contains a data **table**. Tables are lists of **tuples** of **attribute** values. Relations are replicated across hosts.

Relations may be **static**, that is, the table is stored in a disk file, or **dynamic**, that is, the table is computed by a program each time it is accessed. Dynamic relations provide feedback about conditions of resource utilization on the hosts. For example, the output from programs such as *ps* in Unix, which lists characteristics of current processes, can be used to derive attribute values in a table. The output of such programs can be filtered through simple programs (such as *sed*, *grep*, and *awk* on Unix) to convert it into a standard format. Dynamic relations will have tables with different values on each host, but the schemes will be identical on all hosts that have replicas.

Some configuration data are universal across a site, such as host names and address, information about users, and shared disks and printers. Even disparate families often share common ancestry (such as Unix V7) and therefore contain similar mechanisms for configuration. Within a software family, many of the programs, administrative procedures, and configuration data are the same.

**System Administration Language** (**SAL**) is a simple command and query language for **SAT**. It combines command-language features, such as those found in *sh* (Unix) or *DCL* (VMS), with database query features, such as those found in relational databases [Sch77]. The command-language part contains fancy components such as associative arrays and functions. **SAL** is a compiled language; it can be linked with ordinary programs to perform tasks beyond the scope of the language.

## 4 Examples of SAL

We present one example for each of the aspects of the problem of administering a large site. Our purpose is to stress semantics, not syntax; comments prefaced by **$** explain details that might not be obvious.

### 4.1 Configuration Files

At the simplest, a configuration file may be built by a query to the database followed by formatting the result. We present a more complicated example: to create `/etc/crontab`, a file that controls when periodic background tasks should run. Some of this file is constant for all hosts, but other parts depend on the family or other aspects of the host involved. The following program is intended to run on each host.

```
    constant TimeMaster = "j" $ a machine that has a reasonable clock

    @HostsDef.sal $ bring in the declaration of Sal table Hosts
    put FullHostName, OSType, Networks, Console
        into Hosts  $ table name
        from :"Hosts"  $ local replica
        where FullHostName == fullhostname() $ just the entry for self
    endput

    tuple H of Hosts; foreach H in Hosts do $ should be just one
        reopen(stdout,"/tmp/crontab") $ output goes there temporarily

        $ part that applies to all hosts
        echo('
# This file is automatically generated and should not be edited.
0  0 * * *          root   runrand 60 DoAdminReConfig
30 4 * * *          root   cleanup
')
        $ hosts connected to uucp lines
        if H.Networks ~= "uucp" then $ "~=" operator is "contains"
            echo('
10 3 * * *     root   /usr/lib/uucp/uuclean
10 * * * *     uucp   /usr/lib/uupoll
')       endif

        $ Hosts with paper consoles
        if H.Console ~= "paper" then
            echo('
1,31 * * * *   root (echo -n \'    \'; date; echo -n ^M;exit 0) >/dev/console
')       endif

        $ Hosts of the sun family do not keep time very well.
        if H.OSType ~= "sun" then
        echo('
0,10,20,30,40,50 * * * *  root  runrand 2 getdate {TimeMaster}
')      endif
```

```
        $ copy temp file to config file
        close(stdout)
        system('cp /tmp/crontab /etc/crontab')
    end
```

This example shows many of the essential features of **SAL**. A programmer writes a command script in this language. Such scripts typically begin by extracting data from a relation into a **SAL** variable (by the `put` operator). Then the script iterates over the tuples in that variable (`foreach`) using their attributes in conditional statements.

Complex attributes are represented as strings. For example, the `Networks` attribute in the example above contains a list of substrings; it is easy to check for membership with the `~=` operator.

## 4.2 Keeping the File System Coherent

One aspect of coherency is making sure that particular files exist and have reasonable permissions. The following code segment makes sure that all users belonging to group `archive` can modify all files and directories in the anonymous-ftp archive without resorting to 'super-user' permissions. This set of commands, like most coherency checks, is run periodically. The following script should run on all hosts. It uses a dynamic relation `Files`, which describes all files in a given subtree.

```
    @FilesDef.sal
    put FullPathName, Type
        into Files
        from :"Files":"/u/ftp/archive" $ select subtree: /u/ftp/archive
        where FullPathName ~= "^/u/ftp/archive/"
            Group != "archive" || Owner != "root" ||
            (Type == "directory" && Protection != 0775) ||
            (Type != "directory" && Protection != 0664)
    endput

    tuple F of Files; foreach F in Files do
        system('chown {F.FullPathName} root')
        system('chgrp {F.FullPathName} archive')
        if F.Type == "directory" then
            Cfunc chmod(string, integer)
            chmod(F.FullPathName, 0775)
        else
            chmod(F.FullPathName, 0664)
        endif
    end
```

## 4.3 Distributing Software

When a new version of software is developed, it must be tested and then distributed to all machines. The example below shows how a new version of *mumble*, a fictitious software package, might be distributed. We use associative arrays similar to the tables of SNOBOL4 [GPP71]. These arrays can be indexed by any string expression and can contain any number of elements.

```
    @HostsDef.sal
```

```
put FullHostName, MachType
    into Hosts
    from :"Hosts"
endput

function CurrentlyUp(string name) : Boolean
$ test if machine 'name' is up
    string reply
    reply = pipe('/etc/ping {name} 1 1');
    return(reply ~= " 0%") $ packet loss statistics
endfunc CurrentlyUp

tuple H of Hosts; foreach H in Hosts do
    Boolean Seen{20} $ associative array with about 20 expected entries
    if ?!Seen{H.MachType} then  $ not in associative array
        if CurrentlyUp(H.FullHostName) then
            $ H.FullHostName is an example of H.MachType
            $ copy source to chosen host
            system('rcp -r /src/mumble {H.Host}:/tmp/mumble')
            $ compile and copy results to all relevant machines
            system('rsh {H.Host} "cd /tmp/mumble; make install; \
                cd /; rm -r /tmp/mumble"')
            Seen{H.MachType} = true
        endif $ it is up
    endif $ we have not seen its type before
end $ for each host
```

## 4.4 Monitoring Resource Usage

The following code prints a list of accounts that have been inactive on all hosts for 6 months or more. Our example uses the following relations:

```
Hosts       with attribute FullHostName
Users       with attributes Uid, LoginName
LastLogs    (dynamic) with attributes Uid, TimeOfLastLogin

@HostsDef.sal
put FullHostName
    into Hosts
    from :"Hosts"
endput

tuple H of Hosts; foreach H in Hosts do
    @LastLogsDef.sal
    put Uid, TimeOfLastLogin
        into LastLogs
        from H.HostName:"LastLogs" $ Remote query
    endput
    tuple L of LastLogs; foreach L in LastLogs do
        integer LastTime{2000} $ associative array, global scope
        if !?LastTime{'{L.Uid}'} $ not seen this user at all
            || (LastTime{'{L.Uid}'} < L.TimeOfLastLogin) $ or only earlier
            then
                LastTime{'{L.Uid}'} = L.TimeOfLastLogin
            endif
        end $ foreach L
    end $ foreach H

    Cfunc time(integer) : integer
    integer SixMonthsAgo; SixMonthsAgo = time(0) - 60*60*24*30*6

    @UsersDef.sal
    put Uid, LoginName into Users
        from :"Users"
    endput
    tuple U of Users; foreach U in Users do
        if !?LastTime{'{U.Uid}'} $ has never logged in
            || LastTime{'{U.Uid}'} < SixMonthsAgo
        then
            echo('{U.LoginName} has not logged in anywhere within 6 months.\n')
        endif
    end $ foreach U
```

## 5 Database Design and Implementation Issues

**SAT** enables administrators to abstract the information about the configuration of the site in such a manner that

- The information is in a form usable by all hosts at the site.
- The information is protected from accidental corruption by bad edits.
- The information is readily available for more than just administrative uses.

These goals are achieved through the mechanisms provided by **SAT** coupled with appropriate organization and policy imposed by the administrator.

### 5.1 Modifying and Creating Schemes and Tables

We use a **scheme editor** to create a new scheme. It prompts for all relevant information, formats the resulting scheme into an ascii file, and invokes an ordinary text editor on the file. When the editor exits, the file is parsed back into internal form and the scheme is updated. The same program is used to modify existing schemes.

Schemes contain the following information.

- Whether the relation is static or dynamic. (If dynamic, the necessary program is found in a predictable place.)

- A human-readable description of purpose of the relation.
- A file that defines classes of users for specifying access rights.
- A file that defines classes of hosts for specifying distribution.
- User classes specifying who has owner, modifier, appender, and reader rights over the relation.
- Whether the relation is maintained on several hosts.
- Host classes specifying which hosts have model, trusted, copy, and access replicas of the relation. These concepts are discussed later.
- Whether to check the data beyond simple data typing every time the table is modified. (If so, the necessary program is found in a predictable place.)
- Whether to run a report program after each modification to the table. (If so, the necessary program is found in a predictable place.) A report program can regenerate configuration files from the relation when it changes. It can also assist in distributing the relation across boundaries between disparate networks.
- The attributes, including their name, human-readable description, and type. Available types are string (newlines not allowed), text (newlines allowed), Boolean, pattern (string with constraints), real, and integer (with an optional range).

There are several programs for data manipulation. The **table editor** converts the data in a table into an ordinary text form, calls a text editor, then parses the result back into internal form. It checks that the result is consistent with the scheme and invokes the check program, if there is one. Other programs are used for displaying tables (in either human- or machine-readable form) and appending, modifying, and deleting individual tuples.

### 5.2 Access Control

Administration is not the responsibility of one individual; it is usually shared by several staff personnel. It is necessary to enforce policies that restrict access. We satisfy this requirement by access control lists on relations.

As mentioned above, access control lists are specified as a classes of users. There are two predefined classes: **Anybody** and **Nobody**. **SAT** schemes specify the class that is permitted access of four kinds: **owner, modify, append**, and **read**. The owner has full rights to modify both the scheme and the table. Users with modify right may insert, delete, or modify tuples. Users with append right may insert tuples. Users with read right may inspect the scheme and the table. Owner right implies all the others; modify and append right imply read right.

### 5.3 Data Integrity

Every attribute value in **SAT** can be checked to see if it is from an appropriate domain. **SAT** uses two levels of data-integrity constraints. The first level is the type mechanism in the scheme. Strings can be restricted to be newline-free (type `string`) or to conform to a given regular expression (type pattern). For example, a valid DECNET hostname matches a pattern that specifies a letter followed by up to 5 numbers or letters:

```
"^[A-Z][A-Z0-9]?[A-Z0-9]?[A-Z0-9]?[A-Z0-9]?[A-Z0-9]?$"
```

Integers can be restricted to lie in a given range. When an attribute of a static relation is modified, the modification is accepted only if the new value matches the type check.

A second level of data-integrity constraint is provided by programs that check data for global consistency. The scheme for a relation may specify that a checking program be invoked whenever the relation is updated. This program (often written in **SAL**) can enforce such rules as making sure that an account name is not used twice in the Users relation. If the program reports an error, the updates are discarded.

## 5.4 Distributing Relations

Each relation is stored in a separate directory, which contains the scheme, the table, a timestamp, occasionally a lock file, and ancillary programs. The scheme specifies on what hosts to store replicas of the relation. For example, commonly needed relations such as Hosts and Users are likely to have replicas on all hosts. Replicas are categorized as follows.

- **Model replicas** are intended for hosts that have adequate disk resources to hold a replica and are secure (in the sense that ordinary users do not have special privileges). Only model replicas may be directly modified (either in the scheme or table). These changes percolate to other replicas in a manner described shortly.

- **Trusted replicas** are intended for hosts that have adequate disk resources to hold a replica and that are secure (in the sense that ordinary users do not have special privileges).

- **Copy replicas** are intended for hosts such as workstations that have adequate disk resources, but whose ordinary users have enough privilege to compromise the integrity of the data.

- **Access replicas** are intended for hosts such as workstations that do not have large local disk resources but can remotely mount a model, trusted, or copy replica. The host from which they remotely mount the replica must be in the same family, since the associated programs are mounted as well.

For convenience, we will speak of model hosts, trusted hosts, copy hosts, and access hosts with respect to a given relation.

Some hosts have no replica of the table at all. Programs on these hosts that need to read the table must send requests to other hosts (preferably model or trusted). Even hosts that have replicas may send requests to other hosts. For example, dynamic relations may have different values on other hosts. These accesses may easily cross family boundaries because scheme and table format is consistent across all families.

To maintain data consistency, a program to distribute the relation is executed after any change to its scheme or table (which can only occur on a model host). This program is also run periodically to make sure that model and trusted hosts that were down during an update eventually get good data.

The distribution algorithm has two components: gossip and transfer. The gossip phase, based on epidemic algorithms [DGH+88], seeks to inform all hosts that the relation needs updating. It spreads out from the model host on which the changes were made, informing a randomly selected small subset of the other hosts. These gossip messages are timestamped with the date the relation was modified. Each host that hears such a message forwards it in a similar way but does not forward redundant messages.

The transfer phase proceeds in stages. First, model hosts pull the update from the model replica that was changed. Then trusted hosts pull the update from any model replica. Then copy and access hosts pull the update from any model or trusted replica. (Access hosts pull the replica in order to execute the associated report program, if any. In this case, the replica itself is not transferred.) Each replica is timestamped; hosts will only pull a newer replica than what they currently have.

Pulling a replica brings the scheme, table, and source for all programs in the relation's directory. The transfer preserves modification dates. After these files have been transferred, *make* [Fel79] is invoked to rebuild any executables that depend on updated source code. The report program, if it exists, is then invoked.

These actions take a few minutes to finish; meanwhile, stale replicas are still available on hosts that have not yet pulled new ones. We do not provide the same level of consistency control that a transaction-oriented database would because we do not have the same requirements. However, the way locks are acquired (discussed shortly) assures that no model replica is readable until it has been updated.

## 5.5 Locking

We use locking to protect against undesired simultaneous access to a single relation. There is only one kind of lock: exclusive. A relation may be locked by any user with `modify` or `append` right.

The purpose of locks is to prevent access to the relation while either the scheme or the table is undergoing modification. It is quite rare to modify the scheme, and such action often requires that the table be modified to remain consistent with the scheme. A lock may be manually acquired and later released by users who need to make major changes. The lock is usually placed only on the model replicas (and is therefore inherited by access replicas referring to those models). It is possible to lock trusted and copy replicas as well.

Dynamic relations need not be locked when their data change, because they are not associated with a physical table. Static relations should be locked when their tables are modified. The **SAT** package provides routines for inserting, deleting, and modifying tuples in static tables. These routines lock the model replicas for the duration of the change and force a distribution at the end.

Locks are implemented by creating lock files in a majority of model replicas. An attempt to read a locked replica (either model or access from a model) is blocked. Both **SAL** and the other programs for accessing data have options for limiting how long they are willing to wait until the relation becomes unlocked, and they are capable of reading locked relations, although such access is risky.

In order to distinguish locks held by different individuals acting under the same privileged account (such as `root`), all operations involving locks refer to an environment variable `SATID` that may be set to distinguish such individuals.

## 5.6 Implementation

We considered implementing **SAL** either as an interpreter or a translator. Interpretation would make **SAL** program development easier, and small queries could execute without a compilation delay.

On the other hand, we found that a translator was easier to write and had other benefits: **SAL** programs execute faster (interpreted scripts are just too slow), and we have the option of linking in routines written in other languages to handle tasks for which **SAL** is not suited. Compilation time turns out to be acceptible. The **SAL** translator produces code for the C programming language [KR78], which is the language most used at our site for systems programming.

Each relation is stored in a separate subdirectory of the **SAT** root directory. That subdirectory includes the scheme, the table (if it is static), and programs with well-known names for generating dynamic tables, checking consistency, and reporting changes. Ordinary users are forbidden access to these directories.

A daemon runs on all hosts to respond to requests to read and modify relations stored on that host and to set and release locks.

The program that is called when a dynamic relation is accessed is presented enough information (in particular, the attributes and selector clauses in the **SAL** `put` operation) to limit the amount of information it generates. For example, the following **SAL** program generates the names of all the files whose ancestor is `/usr/jones`.

```
@FilesDef.sal

put Name into Files
    from :"Files":"/usr/jones"
        $ the argument "/usr/jones" restricts the query
endput

tuple F of Files; foreach F in Files do
    echo('{F.Name}\n')
end $ foreach
```

The access need not run `stat`, since the `Size` attribute is not selected, nor need it search the entire file tree, since data-set selector `/usr/jones` specifies a prefix.

## 5.7 Using SAT and SAL on Other Families

**SAT** and **SAL** were designed to be portable to any Unix family. The ideas underlying these tools can be used in any family that has

1. TCP/IP-like networking.

2. The ability to have one program start the execution of a second, and when the second has finished, to resume the first where it left off.

3. The ability to redirect the output of a command at least to a disk file, preferably directly to another program.

## 6 Proper Use of the Tools

When a tool is first developed, it is hard to predict what will constitute stylistic use. However, we can comment on a few aspects of administration and discuss how the database ought to be set up and how it ought to be used.

All relations should use the same model hosts. There should be only a few model hosts. Most of these, usually all, are updated every time an edit occurs. This set of model hosts should span more than one family. Otherwise, while the site upgrades to a new software release for the critical family, there may be no available model hosts.

Data should be stored in a general form, even if applications then need to convert to a specific form. For example, string data should be in mixed case, even though some applications may require upper-case only.

Attribute semantics should be carefully chosen. It is unwise to read too many consequences into a particular value. For example, the fact that a host has a disk should not imply that the disk contains user files. At the other extreme, it is also unwise to proliferate attributes that represent trivial facts that change frequently. Our example above that builds the `/etc/crontab` configuration file assumes a `Console` attribute in the `Hosts` relation. This attribute is perhaps unwise.

If an attribute has a limited range of acceptable values, the schema should be as restrictive as possible with respect to type. This policy will prevent failures due to misspelled string attributes.

There should be static relations for hosts and for users. These relations are the most heavily used. The `Hosts` relation indicates all information that distinguishes one host from another, including internet address, hardware and software family, and attached peripherals. The `Users` relation gives the full name, login name, user identifier, group membership, and other information about each user. Passwords should be stored in a separate relation (`Passwords`) that links user identifier to the encrypted password. This relation should not be readable by ordinary users.

Dynamic relations are valuable to describe the file system (`Files`) and the current resource use on each host (`LastLogs, DiskSpace, Processes`).

Static relations may be updated in various ways. Only the `passwd` program should be able to update `Passwords`. Certain staff members may be allowed to append to `Users` and `Hosts`. More experienced staff may modify or delete from those tables. Very few are allowed to modify schemes.

One rule worth following is that configuration files on a host should be modified only by running a SAL script on that host. Our example above of building the `/etc/crontab` configuration file follows this rule. The alternative, generating configuration files on a model host and distributing them, is slower, since it is performed in serial, and it violates the security principle that mistakes made by programs on one host should not interfere with behavior elsewhere.

The algorithm we use for distributing updates can miss a site. Sites that are down during an update are not updated. Therefore, each model, trusted, and copy host should periodically refresh its replica from a randomly chosen model or trusted host. The **SAT** package includes a program that selects a random helper, checks if it has a more recent replica of a given relation, and if so, pulls it to the local site. This program should be run for each relation, but not too often (perhaps a few times every day).

## 7 Related Work

Very few projects in the literature are related to our approach. The ROSI project [Kor86] treats all features of a user's environment as a relation. One feature of ROSI is relations with side-effects. For example, the printer queue is such a relation. A user may append the tuple that represents a data file to be printed to that relation. A background daemon monitors the relation, printing the file tuples inserted there and deleting the tuples when finished.

We use Unix programs such as *awk* [AKW79] to filter the results of other programs into a format acceptable for dynamic attributes. This method is reminiscent of the Awk-as-glue approach [VW86] that uses *awk* to transform data from one format to another. Our project also has similarities to the goal of the *make* program [Fel79], which provides recipes for reconstructing files given precursors. In our case, we reconstruct configuration files given relations.

Yellow Pages [Sun86] is a small distributed database manager used to distribute a few important configuration files between hosts. However it is not very general purpose, allowing only files with formats similar to the Unix password and host files to be used as input data. No data checking facilities are available. Distribution of databases is even more simplistic than in **SAT**.

*Rdist* [RDI86] is a program that helps an administrator maintain identical sets of software within a family. The administrator prepares three lists: all sub-trees of the file system to be distributed, sub-trees or files in the first list that should not be distributed, and to which hosts they should be distributed. *Rdist* checks the modification date and size of each selected file on each target host and updates if required.

*Config* [LK86] is a program that allows the conditional configuration of Berkeley-derived operating systems. The administrator prepares a list of peripherals and options that the operating system should support. *Config* then creates a set of commands to customize the operating system.

The University of Rochester uses a different approach to centralize system administration [Ond89]. Their approach is to provide support for departmental sites by defining each site's environment in a manner that meets the needs of the site but is similar enough to other sites that configuration files from one site will work for another. They have a central staff of system administrators who manage the shared configuration for all subordinate sites. Consistency in the configuration is aided by having each type of configuration task done by one person. They also have developed remote system monitoring tools, which send daily reports about the sites they help to maintain. Their method requires each site to have a local administrator who takes care of such site-dependent tasks as setting up new accounts.

## 8 Status

We have developed **SAT** under descendents of Berkeley Unix. Our installation contains members of three major families: Vax, Sun, and Sequent. Within the families are subfamilies (workstation, server).

We have dynamic relations for disk space and the Unix error log, and static relations for users, groups, passwords, hosts, and the message of the day. So far, we keep all relations on the same three model hosts, one of each family. We have no family-specific relations.

**SAL** has been implemented and heavily tested. We are generally pleased with the syntax, but it is possible that some changes will still be made.

The examples given above are working programs (except insofar as they may refer to relations we have not yet implemented, such as `LastLogs`). We have rewritten *passwd* to use the `Password` relation, which has a reporter program that rebuilds `/etc/passwd`.

We have implemented and tested the scheme and table editors and programs for reading, modifying, distributing, and manipulating locks on relations. Distributed locks are working.

We hope to expand the database soon to encompass another group of machines: AT&T 3B1's and 3B2's, which run different versions of System V.

We intend to distribute our software once it becomes stable at a nominal charge.

## 9 Acknowledgments

David Herron helped to develop an earlier package, which greatly influenced the design of **SAT**. We would also like to thank Eric Herrin and Ken Kubota for their many helpful comments and suggestions. The regular-expression package we use in **SAT** was written by Henry Spencer at the University of Toronto. We also use an environment-variable package written by Maarten Litmaath. We wish to thank both of them for making their packages available to the USENET community.

## References

[AKW79] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk—a pattern scanning and processing language. *Software—Practice and Experience*, 9(4):267–280, April 1979.

[DGH+88] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *ACM Operating Systems Review*, 22(1):8–32, January 1988.

[Fel79] S. I. Feldman. Make: A program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.

[GPP71] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The Snobol 4 Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, second edition, 1971.

[Kor86] H. F. Korth. Extending the scope of relational languages. *IEEE Software*, pages 19–28, January 1986.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1978.

[LK86] S. J. Leffler and M. J. Karels. Building berkeley unix kernels with config. In *4.3 BSD - UNIX System Manager's Manual*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, April 1986.

[Ond89] Denise Ondishko. Administration of department machines by a central group. In *Proceedings of the Summer 1989 USENIX Conference*, pages 73–82, June 1989.

[RDI86] rdist(1). In *4.3 BSD - UNIX User's Reference Manual*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, April 1986.

[Sch77] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[Sun86] Sun Microsystems. Sun network services on the vax running 4.3bsd. In *4.3 BSD - NFS System Administration Guide and Sun Network Services*. Mt. Xinu, Inc., 2560 Ninth Street, Berkeley, CA 94710, June 1986.

[VW86] C. J. Van Wyk. Awk as glue for programs. *Software—Practice and Experience*, 16(4):369–388, April 1986.

# Automated Account Deactivation and Deletion

Don Foree
foree@mcnc.org

Margaret Tiano
mac@mcnc.org

MCNC
P.O.Box 12889
Research Triangle Park, NC 27709

## 1. Introduction

As networked computer systems increase in number of machines and complexity of interconnections, previously simple system maintenance tasks often expand into unmanageable time-sinks. Overloaded operators and administrators begin to ignore time-consuming tasks that can be postponed in favor of shorter or more demanding problems. When a new user appears at the computer room door with an account request form, the task of establishing an account is obviously one which cannot be postponed for long without affecting productivity. But when the same person leaves three years later, it is too easy to postpone the deletion of the account and disposing of files, even though we, as system administrators, know that this leads to wasted disk space, cluttered administration files, wasted cpu cycles and wasted staff time processing this extraneous information. The nature of our company, a consortium of industrial and academic institutions involved in microelectronics design and research, is such that we often have visiting scientists and students who are on our staff for varying lengths of time. During these sometimes brief stays, however, relatively large quantities of computing resources are often consumed. By the time account deletion was noted as a problem area, the operator whose duty it was to delete accounts had a stack of almost 100 accounts that were in need of deletion. By writing some scripts to automate the deletion procedure she was able to eliminate the backlog and keep up with current load.

Our account deletion process consists of two parts. First an account is **deactivated** and then after a suitable delay period during which supervisors decide how best to dispose of the former user's files, the account is **deleted.** Both procedures are automated by KornShell [1] scripts.

## 2. Account Deactivation

The account deactivation script, **deactusr,** is relatively simple. **Deactusr** requires one argument, the login name of the user to be deactivated. In addition there are three options that will be discussed below. Since the script alters the file **/etc/passwd** it must be run by root. The major functions of **deactusr** is to replace the user's login shell with a script that simply puts a message on the user's screen that his account has been deactivated and to change the user's home directory permissions to 000. The message lets the user know the reason for his/her inability to log onto the system. The change in permissions has the dual effect of disallowing access to the user's files by both the user and anyone else who might have been dependent on the deactivated user's files. Persons finding that necessary files are no longer available inform the support staff who can then take appropriate action before the files are deleted from the system.

A -N option allows a mail message to be sent to a specified person (usually the user's supervisor) indicating that the user's account has been deactivated and giving a list of machines on which this has taken place. The message asks the supervisor to determine an appropriate disposition scheme for the deactivated user's files. A -R option allows a login name to be specified in whose **~/calendar** file a notice will be placed as a reminder to delete the account in question at some future date. (We allow a period of three

[1] Morris Bolsky and David Korn, ***The KornShell Command and Programming Language***, Prentice Hall, 1989.

weeks).

At MCNC a person might have accounts on from one to over forty machines. The **deactusr** script checks each MCNC machine [2] and makes a list of those on which the user has accounts [3]. An option (-e) to **deactusr** allows the administrator running the script to edit the list before accounts are deactivated. This allows the accounts to be deactivated on a subset of the machines on which a user has accounts.

## 3. Account Deletion

The account deletion script, **delu**, requires a single argument, the login name of the account to be deleted. It has a single option (-e) which, as in **deactusr**, allows the operator to edit the list of machines on which accounts are found. A copy of **delu** resides on each machine in our system. Under normal circumstances, it is only run from one of the larger computers which has a large amount of disk space available in /usr/tmp in which to collect the user's files from the other machines. If network problems exist, however, it may be run on individual machines which may be cut off from the rest of the network. Like **deactusr, delu** requires that the script be run by the root user. **Delu** relies on four shorter scripts which reside on each machine. This makes **delu** more manageable and allows the use of the shorter scripts by themselves.

**Rmpasswd** uses **vipw** to remove the entry in /etc/passwd of the user whose login name is passed as the single parameter.

**Rmquotas** uses a locally written program to remove quota information for the user passed as the single argument from all filesystems on which they exist.

**Rmuserid** requires two arguments, a login name and a file name. It removes all instances of the login name from the file. It is designed to work on files like **/usr/lib/aliases** and **/etc/group** where a login name is delimited by commas, colons, spaces or end of line.

**Roundup** makes a complete check of all host filesystems. A list is made of all directories owned by the login named passed as a parameter. A second list is made of all non-directory files owned by the same user. These lists are reduced to the smallest possible list of directories and individual files that must be moved in order for every directory and file owned by the user to be included. The completed lists and all of the files included in the lists are then copied to a location passed as another argument to the script (normally a location in /usr/tmp on the local machine). **Roundup** only makes copies of files, it does not move or delete any. Its function is to find all of those files that exist outside of the user's home directory. The disposition of these files requires operator intervention. Working from the lists created by **roundup** and sometimes conferring with owners of directories where these stray files are found, the operator will change ownership of some of the files, move some to a public location and simply remove others.

**Roundup** is the most time-consuming portion of the account deletion process. On machines with large amounts of disk and user load, it can take as long as 20 minutes to check every file for ownership. For this reason, our operators who normally use **delu** usually run it in a separate window on their workstations, attending to that window when necessary.

**Delu** itself checks each machine in the system for accounts by the user being deleted. The operator is shown the list of machines on which deletion will take place and allowed to edit that list if the -e option was used. The operator will be required to know the root password for each of the machines in the list. Knowing the list ahead of time allows the operator to be sure that these passwords are known. A staging directory is then made in **/usr/tmp** to hold all of the user's files from all system machines.

Next for each machine in the list, a remote shell is started, the user's mail spool file, if any, is moved into his home directory and the user's entire home directory is moved via **tar** to the staging area on the machine on which **delu** is being run. At MCNC this file transfer is performed via Freedomnet [4] but it

[2] At MCNC machines are grouped into various overlapping *hostclasses* which are updated each time a new machine is added to our system. The scripts discussed here always search through the *hostclass* =all which includes all MCNC machines.

[3] The function of locating a user's accounts on all machines was found to be so useful that a separate script was developed to provide quick access to this information.

could be performed via **rcp**. In addition, any stray files found and collected by **roundup** are transferred along with the list of stray files to the staging area on the home machine. The home directory and mail file are deleted from their original locations, but the stray files are not (see **roundup** above). If all of the file transferring goes well, the user's name is removed from **/etc/group** and **/usr/lib/aliases.** His/her quotas are removed and finally his line is deleted from **/etc/passwd** on the remote machine. In the staging area, the files collected from a given machine are untarred before the account is deleted from the next remote machine. When all of the remote machines have been completed, the deletion process is performed on the staging machine. At this point all of the deleted user's files are in a staging area under directories named for each machine from which the files were taken and all traces of the user are gone from all machines except the staging machine which holds all of these files. The operator may then dispose of the stray files found by **roundup** on each remote machine and **tar** the entire contents of the staging directory onto tape for archiving.

## 4. Conclusion

By automating the account deletion process, we make several gains. First, accounts are deleted in a timely fashion saving computer and staff resources. The actual time involved in deleting an account is cut allowing operators to deal with other problems. Finally, the deletion process is defined by the scripts. Once the scripts are tested and implemented, the room for administrative error is greatly reduced. If a new aspect of account deletion is required, the script is changed once, instead of requiring the operator to learn a of a whole new set of procedures.

[4] Freedomnet is a software subsystem developed by the Research Triangle Institute, R.T.P, NC, that when installed on physically interconnected, heterogeneous computer systems, allows a user to access the resources of all of the computers in the network in a transparent manner.

# Mkuser - or how we keep the usernames straight

*Gretchen Phillips* - SUNY@Buffalo
*Ken Smith* - SUNY@Buffalo

June 1989

## 1. Introduction

The State University of New York at Buffalo has an ever growing Unix based computing environment. The primary instructional Unix machines now include a VAX 11/785 and a Sperry 7000/40 both running Berkeley 4.3, as well as an Encore Multimax running Umax 4.2 Release 3.3. Additionally, Sun workstations now proliferate on the campus. Administration of any number of Unix machines becomes a headache when username administration comes into the picture. Additionally, with NFS filesystems, users must have consistent uids along with usernames (where consistency is for the convenience of the user).

The total number of unique usernames in the timesharing environment is over 1200 and workstation accounts number about 500. When Unix timesharing was the primary location for users, and system administration had only one point, management of usernames and uids was relatively straightforward. Initially, a simple c-shell script was sufficient for adding users (when we had only one machine). This grew to a C program that could take a batch file for processing instructional accounts and would check existence of duplicate usernames and uids on the other timesharing hosts. This system fell apart when workstations began to proliferate on the campus. There was more than one point for system administration; individual departments or schools hired system administrators, individual workstation owners could add and delete users, and although cooperation was desired, no tool was available to maintain usernames over the entire network in a consistent fashion. To this end, we developed our newest incantation of *mkuser*. It has a daemon uid and username server and database manipulator, as well as a client program for manipulating appropriate system and user files on hosts throughout the network.

## 2. Philosophy of *mkuser*

The philosophy of usernames, in general, isn't a complicated thing. People need a method of access to resources and some place to store their work. Managing this can be as simple as one username where everyone shares, cooperates and never makes a mistake, or as complicated as one (or more) username for each user. We settled on one username per user. Later, we additionally settled on one uid per user.

The original script we used for making users was simplistic at best and dangerous at worst. It simply prompted for necessary passwd fields, did some minimal checks and appended to /etc/passwd. The growth of our environment and expansion from one where users were concentrated on a single machine to users spread over four machines (or more), meant that we had to come up with a more sophisticated method for maintaining accounts. Additionally, at the beginning of a semester it could be the case that three hundred new accounts needed to be created. Needless to say, this was an ugly task.

We developed a C program that would prompt for necessary information in an interactive fashion or read a batch file for massive account additions. One essential feature that we added was to have it check for username duplication on the other timesharing hosts. This was possible because all system administration was handled by a limited number of trusted people and the hosts were equivalenced. The reasons for this feature were two fold. It would be a bad thing if two different people ended up with the same username. It would be inconvenient if a single person ended up with two different usernames.

With the additional constraint of a consistent unified uid base and the proliferation of Sun workstations and system administrators on campus, we decided that we must develop some type of uid/username server. With this tool any person who generated a username could be confident that neither the username nor the uid conflicted with another on the network (either within a sub-domain or between domains). It is especially critical within a domain for the purposes of mail delivery when a domain spans several machines. The incantation became *mkuserd* (the server) and *mkuser* (the client).

## 3. Overview

The *mkuser* and *mkuserd* programs are basically straightforward. The client, *mkuser* collects information from the system administrator. This information can come from interactive or batch input. It then makes requests to the daemon, *mkuserd*, based on this information. *mkuserd* in turn responds to transaction requests and sends appropriate information back to the client. *mkuser* then uses this information in creating new accounts. Additionally, *mkuserd* stores information about the user in a centralized database.

## 4. *mkuserd*

The *mkuserd* program runs on one of our timesharing machines. It is the core of this network implementation. The functions of *mkuserd* include: determining if the user has an existing account determining if a username is in use providing unique uids collecting password files from client machines and updating data base maintaining useful information about the user

*mkuserd* can be described as an transaction processor. It takes transaction requests from *mkuser* clients, processes them and returns some information. If a client supplies a user identifier, then if that unique identifier has a username associated with it, *mkuserd* will return the username and uid. If no username is associated with that unique identifier, *mkuserd* returns a null string indicating no username exists. If a client supplies a username, then *mkuserd* will determine if that username is in use. If it is, then *mkuserd* returns the full name of the user associated with it. Finally, *mkuserd* will return an unused uid when requested.

Access to this information is provided through hash tables based on username and unique identifier. This allows efficient access to the data through the two primary key areas. Unused uids are stored in a array and supplied to client programs as uid transaction requests are made.

### 4.1. *mkuserd* database

*mkuserd* keeps information on all accounts that are created and in fact all accounts in the client password files. In particular it keeps records that include: username user-id group-id unique identifier for each user user's full name department information expiration date machine name(s) where user has account
Whenever a *mkuser* client connects to *mkuserd*, *mkuserd* requests a copy of the current **/etc/passwd** file from the client host. It uses this information to update any account and uid information that may have been inserted or deleted "by hand" and reminds the *mkuser* client that additions should be done only by *mkuser*. With current password files, *mkuserd* keeps its internal database up to date. It checks that uids still have the same value that exist in the database. If inconsistencies are found, then it sends an informational message back to the client. The local system administrator is responsible for making the appropriate changes based on the messages sent by *mkuserd*.

### 4.2. *mkuserd* security

Security of the data is provided by assuring that the connection between server and client is run on a privileged port. This prevents random users from connecting to the port and obtaining any potentially sensitive data. Additionally, the transmission of the unique identifier is uni-directional. The client sends this to the server but the server never sends it back. This prevents data from being transmitted should the port be compromised.

### 4.3. *mkuserd* problems

*mkuserd* has some faults. It is a memory pig. Since *mkuserd* runs on a single host, if the host is inaccessible, then *mkuser* cannot be used to generate accounts. It is, however, possible to move the database and *mkuserd* to another host and then reconfigure client configuration files to know the new location of the server without recompiling the client or server programs.

## 5. *mkuser* clients

The *mkuser* client program takes input describing users and connects to *mkuserd*. *mkuser* makes transaction requests from *mkuserd* and uses this information to create local accounts. It creates password file entries and home directories based on the username, uid and gid provided by *mkuserd*, as well as information stored in a local configuration file. The input to *mkuser* can be either interactive or batch. Typically, batch files are created from student registration data where student identifier, full name, course registration and university standing are recorded. These files are processed by *mkuser* and unique identifiers are passed to *mkuserd*. If that identifier is present in the database, the username is returned to *mkuser*. If no account exists, *mkuser* generates a username and then asks *mkuserd* to verify the uniqueness of it. Upon verification of uniqueness of the username, *mkuser* requests a uid from *mkuserd* for the username.

### 5.1. username generation

The generation of usernames on other SUNY@Buffalo machines (VMS and CMS) is based on the unique identifier. This results in usernames of the type v117fpl4 and c999il1t. We hoped to be more generous in our generation of usernames and settled on an algorithm of selecting the first unused username generated based on a combination of the first name, middle initial and last name of the user. This can be overridden with a runtime flag in the interactive mode that will prevent the *mkuser* client from generating a username and will instead prompt for one. This is especially handy when *mkuser* cannot generate a unique username using this algorithm and the system administrator must generate the name using one of those human brain based algorithms.

### 5.2. *mkuser* configuration

*mkuser* consults a local configuration file. Local system parameters, established by the system administrator, are stored in the *mkuser* configuration file. The configuration file is stored in /usr/local/adm/mkuser/mkuserd.conf. Configuration parameters stored here include: location of the server username prefix home directory location quota limits These variables include the location of home directory and quota values. These are based on groups established by the system administrator. This permits *mkuser* parameters to be set by the local administrator. For example, one administrator may have all home directories in a flat directory, say /users, while another may choose to segregate into /users/faculty and /users/student. Default quota values can be set for different groups or invocations of *mkuser*. Local administrators can supply configuration parameters to the username generation algorithm so that usernames can be generated with a departmental prefix. The prefix was not part of the original username generation scheme but was a compromise for those departments who wish to have some distinction among groups of users based on username.

Parameters are read at runtime so they may be changed as the system administrator deems necessary. This is handy when partitions get full and new users need to have home directories in different partitions.

### 5.3. *mkuser* expiration

In addition to adding users to a system, *mkuser* can be used to expire users. A runtime flag specifies that it is a expiration run, rather than addition and usernames are expired from the password file. Expiration dates are stored in the data base. An expiration run will deactivate all accounts that have passed their expiration date. This feature is critical in our environment where there are well defined boundaries on account activity. Expiration dates are only checked on an expiration run so accounts are only removed at the request of the administrator.

## 6. Conclusion

This system does not completely free system administrators of account administration. System administrators still have the responsibility of correcting any problems in their /etc/passwd files. They must still obtain registration lists to feed to the *mkuser* client. They must still run *mkuser* for adding accounts. It does relieve them (or some campus wide administrator) from having to maintain a list of available uids. It does relieve them of having to check to see if a user has an existing account. It does relieve them (for the most part) of having to use some algorithm in their brain for generating usernames. Finally, it does offer some method for expiring accounts based on expiration dates. In the overall view, *mkuser* and *mkuserd* can make the administration of a distributed environment, by a group of distributed administrators, a less frustrating task.

# Disk Space Management Without Quotas

Elizabeth D. Zwicky
SRI International
zwicky@spam.istc.sri.com

July 28, 1989

Several years ago, we had 600 megabytes of disk space, which we thought was a lot of space but somewhat cramped because our users were disk hogs. Now we have 14 gigabytes of disk space, which is a lot of space but somewhat cramped because our users are disk hogs. In the interim, we started using the Network File System (NFS) and stopped being able to use the traditional UNIX quota system.

Quotas interact badly with NFS in several ways:

1. If you are running quotas, all file systems are checked at login. This means that every machine with a file system on it must be up in order for people to log in. This problem can be surmounted to a certain extent by using soft mounts, interruptible mounts and/or an automounting scheme, if your machines all support these. Ours do not. It can also be avoided by linking **quota** to `true`, and making users do something else to actually get a quota listing.

2. Not all machines actually manage to propogate quota warnings to users if the users are accessing their files over NFS. From a Pyramid, attempting to write a file to an NFS-mounted file system on which you are over your hard quota results in silent failure.

3. Using **quota** to check your quotas does not usually give correct results over NFS. It may claim that you do not have quotas set on filesystems on which you are over quota; it may show quotas but get important facts wrong (for instance, the amount of time you have left before your soft quota is enforced).

4. Not all machines which run NFS run the normal quota system.

The traditional quota system is also difficult to administer when there is a large user base, because it requires that a quota be set individually for each user. **edquota** does provide options to ease this process by making a model user and copying that to other users, but this is neither easy nor flexible. By default, users have an infinite quota, which is a little more trusting than we like to be.

Because of these difficulties, we are unable to run the quota system and have quotas enforced by the kernel. Our solution has been to administer quotas as necessary when file systems run out of space. We use a Yellow Pages (YP) map to distribute information about what quotas people have; the makefile that propagates changes to the data within YP finishes up by copying the database as a file to those of our hosts that do not run YP. Quotas are specified in kilobytes, using -1 to indicate infinity. A quota can be assigned to a group or to an individual, and on a specific file system or all file systems.

When we wish to enforce quotas, we run a program called "qkill" (the current version is a perl script, written by J. Greely). `qkill` determines the type of system it is running on, and uses the appropriate program to get a list of all users and their disk usage. As a speed improvement, it uses a locally modified version of `quot` if it is running on a Sun server. For each user that appears in the resulting listing, it checks the quota database for an individual quota for the file system it is running on, an individual quota for all file systems, a group quota for the user's login group for the file system, or a group quota for all file systems, and setting the user's quota to the first one of these it finds. If no quota is specified, it assigns a default quota of 1 megabyte, which is our normal quota for guest accounts and undergraduate students.

Using command-line options, you can specify whether it should simply display the list of people who are over quota, in descending order by the percentage they are over quota, or whether it should send mail to those users. You can also specify how much over quota users must be to receive mail; our standard is to send mail to users who are using twice as much disk space as they are supposed to. Most users will comply with the message, although some either fail to read their mail, or ignore it.

Since qkill provides no way to actually force users to immediately come within quota, we have developed a policy and a program to clear disk space. The policy is that people who are consistently and ridiculously over their quota get a warning of impending doom, after which we wait seven days, and if they are still over quota, compress all their files. Then we wait seven more days, and if they haven't made it within their quota at the end of the second week, we tar enough of their files to tape to get them within quota, without any particular regard to how useful the files look, and send them a message telling them where to pick up the tape. Once their files have been compressed, people tend to realize that we're serious, and we rarely have to resort to tar.

The program we use to immediately clear disk space is `cleanup` (a C program written by Diana Smetters). It recurses down a directory tree from a specifed point, looking for files that it can reasonably assume are recreatable. For instance, it will remove emacs backup files if they are identical to the current copy of the file; core files; .o files with corresponding .c, .p, .f or .cob files; and LaTeX and Scribe output and log files with newer .mss or .tex files. As an extra level of caution, it generally ignores removable files that have been recently accessed. Command line options allow you to specify what set of things to remove, whether or not to do it interactively, and how many days a file must have been untouched before it can be removed. How much space `cleanup` frees will depend on how cluttered the file system is. On a 245 megabyte file system that has not recently been cleaned, it usually frees between 10 and 20 megabytes. One memorable day we used the local option for use on undergraduate students and freed 40% of the filesystem.

We supplement these with `oldkill` (a shell script which I wrote), which produces a table showing, for each subdirectory of the directory it was started on, the owner of the subdirectory, the number of files not accessed in 365 days, accessed within 365 days but not within 180, and accessed within 180 but not within 90, and the name of the subdirectory. It also has an option to send mail to the owners, listing the names of all the files in each category, and indicating that they should probably be compressed, deleted, or archived to tape.

While it would be nice to have quotas automatically enforced (in particular, the users are more resigned to being fussed at by machines), this system allows us to effectively control disk usage. The programs that it uses are in general simple enough to be run on any UNIX machine, which is an important consideration in our environment, where we are running nearly 10 versions of UNIX. The defaults also greatly simplify administration, since of our average 1,600 accounts, nearly 1,000 are changed every 12 weeks, as the undergraduate student population changes from quarter to quarter. The account installation procedures do not need to set quotas at all; the undergraduate quotas are all handled by the group defaults, which are provided by the instructors.

The one deficiency that our current system shares with the traditional quota system is that it does not handle group directories well. Their disk usage is charged against the owners of the individual files. We are currently working on a new version of `qkill` that allows you to set a quota for a group, as well as for its members, and that takes into account membership in multiple groups. Currently students who are in multiple classes get the quota for whichever class appears first on the university's rosters, which may or may not be the highest quota. Under the new system, quotas for members groups are additive; every user gets a base allowance, plus the group member quota for each group they are in. The group itself may have a separate quota, and all members of the group are notified when the group is over its quota.

# Administering Remote Sites

*Peter Zadrozny*

Electronic Data Systems
Bloomfield Hills, Michigan
(313) 645-4703
peter@edstip.eds.com

*ABSTRACT*

This paper describes the way remote system administration is performed from a central location. It presents the procedures and documentation used to perform these duties and to minimize the presence of a system administrator on the remote site.

## 1. Introduction

The Lab Support Group of the Electronic Document Management Division is responsible for designing, installing and maintaining Local Area Networks (LANs) for production sites. The production sites use the LANs to tackle diverse problems that range from creating and composing Owners Manuals for vehicle groups of General Motors to complete imaging systems for the insurance industry. The networks are basically used to run third party applications such as publishing and illustration systems as well as sophisticated imaging and document management systems. The LANs generally consist of a heterogeneous mixture of Unix based workstations (Sun, Apollo, HP, DEC). Sometimes PCs and Macintoshes are part of the networks and they are mainly used as authoring stations (text/graphic creation). The networks vary in size from just a few workstations to 70 or more.

The remote networks are highly static from the system administration point of view. Once the the system administrator has installed the hardware and software there is very little to be done other than making sure that things are working correctly and trying to anticipate any possible operational problem. Problems regarding applications are handled by a different support group.

## 2. Network Data Book

In order to overcome the hurdle of not being on the site, all the information related to the remote network is kept in the Network Data Book. This book contains a combination of the following forms, when applicable, but not limited to:

- A general description sheet. This sheet provides a general description of the LAN, its geographical location, ownership, usage and other pertinent information.
- A functional diagram sheet. This sheet depicts the types of workstations and their peripherals as well as the software that runs on them. This diagram is used to obtain a fast and general idea of the network.

- A physical layout sheet. This sheet illustrates the location of every node, the electrical connections, telephone and modem lines, and the network connections. The level of detail in the layout can get to the point of showing if the connectors on a thin wire run are crimped or soldered.
- A node data sheet. This sheet contains administrative information such as serial number, service contract, asset and purchase order numbers, installation date and installers name, as well as technical information such as manufacturer, model, node type, memory, host id, internet and ethernet numbers, operating system and version, type of electrical power (conditioned/surge suppressor, etc.). It also contains a list of the attachments (models and serial numbers) of the node such as the console, mouse, keyboard, controllers, expansion boards, etc.
- A disk data sheet. This sheet contains similar administrative information as the node data sheet and detailed technical information regarding every hard disk, such as number of cylinders and alternates, heads, sectors, partitions including sizes and mounting points, etc.
- An application software data sheet. This sheet describes the administrative issues of the application software. This information varies from package to package, but the common points boil down to floating/fixed license, passwords, application directory, special links, application hot line number and contact person, and comments on side effects, special links, and usage of other Unix tools.
- Log sheets. There are two types of logs, the network log and the node log. The later is an account of any changes, upgrades or modifications, service calls, problems and solutions applied, be it hardware or software, of every node on the network. The network log is similar but applies to the whole LAN.
- Checklist sheets. These are hardcopies of the checklists used during installation of the hardware and software.
- Key files. These are printouts of key files such as rc, rc.local, rc.boot, exports, fstab, hosts, printcap, crontab, services, etc.
- A contact sheet. This is a list of key people to this particular site, such as system users and their supervisors/managers, etc. The list includes phone numbers for the site, beeper and home as well as hours/time zone they can be reached.
- A disaster recovery sheet. This sheet describes the various disaster recovery plans based on the possible events, alternate sites and procedures to activate them.
- An operations and procedures sheet. This sheet contains a description of this particular site (explained in the next section). It also describes the backup and maintenance procedures, which includes details as devices, media type and brand, rotation of media, etc.
- A communications sheet. This contains information about electronic mail such as map entries, communication times, modems, etc.

## 3. Operations and Procedures

Although every site is to some degree different, certain things are common among the sites. One of them is that nobody at the remote site has knowledge of root's password. This is only known at the central location. If there is the need of performing operations that require root's ID, special logins are set up to perform only those operations.

Every remote site has a designated *key operator*. This is a person that is responsible for the very few system administration duties that require on site presence. The *key operator* is also the main contact person between the remote site and central support. Any problem that occurs, hardware or software (except production applications) is reported to the central location where it is handled or delegated to the appropriate entity.

In order to prevent problems, the central location periodically monitors the remote systems. Shell scripts are started at various times of the day via crontab (times and contents change from location to location). These scripts invoke the standard administrative and statistics tools provided by Unix (e.g: vmstat, iostat, pstat, df, etc.). The results are either mailed immediately or appended to a file to be mailed later. In some cases the scripts analyze the results and send alarm mail if a certain threshold is reached. Mail messages are monitored at the central location and when a problem is anticipated either the *key operator* is contacted, the remote site is dialed-in, or in grave emergencies a visit to the site is done.

This way of operation is based mainly on UUCP mail which implies that at least one modem is attached to the LAN. This is a potential security gap (as any modem on a Unix system can be). Some sites handle extremely sensitive data where others could care less about the security. UUCP mail is used unidirectionally, from the remote site to the central location and only for system administration. In any case provisions have been taken such that only root@central_location can dial in.

Backup policies again, change from site to site and depend on various factors which include data criticality, number of shifts, amount of data produced/modified daily, weekly, etc. Partial backups are started automatically by crontab and tapes are loaded/unloaded by the *key operator*. If full backups are done more often than monthly, the *key operator* is also responsible for them. A special login is set for such an operation and it executes scripts which are network dependent that will guide the *key operator* while performing the full backups.

Sites that are geographically close to the central location are visited once a month, otherwise they are dialed-in, and a general checkup of the system is performed (log checking, trimming, and the likes). If the visited sites are on a monthly schedule of full backups, they are performed during the visit.

The inevitable rotation of personnel creates the necessity of adding, modifying and deleting users from the networks. This is handled in two different ways. On one the *key operator* uses a special login to perform these operations and the central location is notified via e-mail. The second way is where the *key operator* issues a request via e-mail to the central location which then satisfies the request dialing in to the network.

Disaster recovery plans are devised for every location, and in most cases a partner site is assigned. A partner site is an alternate location that is prepared to handle the operations necessary to satisfy the minimum working requirements of the affected site, thus avoiding a complete shutdown of it. When this cannot be done, be it for uniqueness of application or else, the partner site is the central location. The central location has a copy of the applications used at the remote sites, since they are tested and benchmarked before being installed (or considered, for that matter).

Finally, upgrades and patches to applications and the operating system are performed by personnel from the central location when applicable or by the *key operator* with telephone assistance from central support.

location where it is handled or delegated to the appropriate entity.

In order to prevent problems the central location periodically monitors the remote systems. Shell scripts are started asynchronously at the central location via crontab entries and content change from location to location. These scripts invoke the standard administrative and statistics tools provided by Ultrix 2.0 (vmstat, iostat, pstat, df, du, ...). The results are collected and immediately [illegible] to the main system. In some cases the scripts analyse the results and send mail to [illegible] if a certain threshold is reached. Mail log files are monitored at the central location and when a problem is suspected either the key operator is contacted, the remote site is dialed into or appropriate action is taken via the system's modems.

This way of operating is based on trust on DECT [illegible] which implies that at least one system manager at the LAN [illegible] potential security [illegible] on a remote system [illegible]. Some sites handle extremely sensitive data, where others could take [illegible] about the security. Until [illegible] is used electronically from the remote site to the central location and only for system managing [illegible]. In those cases precautions have been taken such that only the central location can dial in.

Backup policies at all sites range from daily to weekly and depend on various factors which include the criticality, number of users, amount of data produced, modified daily, weekly, etc. Partial backups are started automatically by crontab; full tapes are loaded/unloaded by the key operator. Full backups are done more intermittently; the key operator is also responsible for this. A special login is set for such an operation and it executes scripts which are network dependent, that will guide the key operator while performing the full backups.

Since the key operator's expertise [illegible] the central location are visited once a month, since they are remote, and a general clean up of the system is performed (log checking, trimming, [illegible]). [illegible] remote sites [illegible] a monthly schedule of full backups that are performed during the visit.

The inevitable problems of personnel changes the necessity of adding, modifying and deleting users from the network. This is handled in two different ways. One by the key operator using a special login to perform these operations and the central location is notified via e-mail. The second way is where the key operator sends a request via e-mail to the central location which then satisfies the request adding it to the network.

Disaster recovery plans are devised for every location and in most cases a partner site is assigned. A partner site is an alternate location that is prepared to handle the operations necessary to satisfy the minimum working requirements of the affected site, thus avoiding a complete shutdown of it. When this cannot be done, because of the uniqueness of the application or size, the partner site is the central location. The central location has a copy of the application used at the remote site since they are tested and benchmarked before being installed (or considered) for that matter.

Finally, upgrades and patches to applications and the operating system are performed by personnel from the central location when applicable or by the key operator with telephone assistance from central support.

# Enhancements to 4.3BSD Network Commands

Helen E. Harrison
heh@mcnc.org

Tim Seaver
tas@mcnc.org

MCNC
P.O. Box 12889
Research Triangle Park, NC 27709

## 1. Introduction

The Berkeley networking 'r' commands (*rlogin*, *rsh*, and *rcp*) provide easy, convenient mechanisms for network access to remote machines. To access machine B from machine A, just set up a network access permissions file (*.rhosts*) on machine B and never worry about it again; it will always work. Unfortunately this is an all-or-nothing problem: you must have this permissions file on B to use *rcp* or *rsh* to B. The only choice allowed by the Berkeley authorization scheme is password-free access or no access. Brian Reid, in a guest column in the Communications of the ACM [1], describes computer break-in problems at Stanford which became widespread in part due to the "fundamentally vulnerable" .rhosts access between computers. He concludes that it would be useful "to find ways to permit people to type passwords as they are needed, rather than requiring them to edit new permissions into their permissions files It would be quite reasonable for a system manager to forbid the use of them, or to drastically limit the use of them Programmer convenience is the antithesis of security, because it is going to become intruder convenience if the programmer's account is ever compromised." At MCNC we have made extensions to the Berkeley "r" commands which address these concerns.

MCNC is a consortium involved in cooperative research between academia and industry. These academic institutions are joined by a statewide microwave network. Often our users originate at other sites on networks over which we have no control. Because we could not control the security of remote hosts we decided that our administrators should control whether users on those hosts should be allowed password-free access into our machines. We modified the Berkeley access mechanisms so that a user's individual *.rhosts* authorization file is effective only if he is coming in from a 'trusted' machine. Unfortunately, this completely denies access via *rcp* and *rsh* from untrusted remote sites. Therefore, we also added to these commands the ability to prompt for a user's password when access would otherwise be denied. With this change, *rcp* and *rsh* will prompt for a passwd if access is denied either because the user's local machine is not trusted or because his *.rhosts* file does not include his local machine. Because of our close association with other institutions, we also implemented a mechanism to make it more convenient for a user to manage different login names across administrative domains. A user may set up a user/host mapping on the local machine which will be used as a default when accessing a remote machine. These local mapping files are in some sense an inverse of the remote *.rhosts* files. These three mechanisms work well together but may be used independently. They consist of a set of C library routines and changes to the original Berkeley programs.

## 2. The *Rcom* Library and Its Invocation

The 4.3BSD 'r' commands use the *rcmd*(3X) library routine for remote command execution and the *ruserok*(3X) routine for access authorization. MCNC replaces these with *rcom* and *ruserallow*. The *rcom* routine replaces *rcmd* for establishing a SOCK_STREAM connection with a remote machine to execute a command. A local *rcom* first tries to connect to the remote command execution server, *rshd*, which uses the *.rhosts* mechanism

[1] Reid, Brian. "Reflections of Some Recent Widespread Computer Break-Ins." *Communications of the ACM* 30,2 [February, 1987], 103-105.

for authorization. If this fails, the local *rcom* prompts the user for his remote password and tries to connect to the other remote command execution server, *rexecd*, which uses password authorization. *Ruserallow* replaces *ruserok* in evaluating whether the user has permission to establish a connection. Its permission checks are more strict than those of *ruserok*. For a local user to obtain remote connection permission, the local machine must be among those trusted by the remote machine. Trusted machines are listed in the file */etc/rhosts.allow*. In addition, the user's *.rhosts* file on the remote machine must include the local user on the local machine. For remote user name mapping, the library routine *rname* returns the appropriate login name to use on a specified remote machine. This mapping is done on the local machine with either in an environment variable, *RNAMES*, or a file *.rnames* in the user's home directory, in the same format as the *.rhosts* file. *Rcom* uses *rname* to determine the correct userid to use if it is not explicitly stated. This integrates *rname* easily into programs which use *rcom*.

The *rcom* library routines have been incorporated into standard 4.3BSD commands and daemons. To use the MCNC authorization scheme, *rshd* was modified to invoke the *ruserallow* routine instead of *ruserok* in determining whether to allow a requested connection. Similarly, */bin/login* was changed to call *ruserallow* instead of using its own authorization routine when called from *rlogind*, the remote login server. In this way, access into a machine from *rsh*, *rcp*, or *rlogin* is controlled by the */etc/rhosts.allow* and *.rhosts* files. To work with the new authorization scheme, *rsh* and *rcp* have been modified to call *rcom* in place of the Berkeley *rcmd*. Thus a user will be prompted for his remote password when using *rsh* or *rcp* if he is not on a trusted machine and/or does not have his remote *.rhosts* file set up correctly. By using *rcom*, *rcp* and *rsh* already benefit from the *rname* mapping without any additional effort. *Rlogin*, however, had to be modified to use *rname* directly in order to use remote name mapping between sites, as it still calls Berkeley's *rcmd*.

*Rcom* is built on two other routines which are also generally available. *Rcom_shell* talks directly to *rshd* on the remote machine using a protected network port. It may only be used by the super-user. *Rcom_exec* connects to *rexecd* on a remote machine and passes the user's remote password. It may be used by anyone. These routines are useful if the *rcom* design does not fit easily into your application, for instance, if you want to handle password prompting or error reporting in a special way. In addition, two error routines are available to help diagnose problems. *Rcom_error* returns success or failure according to whether a particular *rcom* return code is an error. *Rcom_perror* is similar to *rcom_error* except that it also prints an appropriate message if there was an error.

## 3. Examples

As an example, suppose user Joe Cool has accounts on the departmental machines 'math' and 'cs' (see Table 1). His login names are jc@math and cool@cs. The math department 'trusts' the machine cs (i.e. math has 'cs' in its */etc/rhosts.allow* file), but cs department does not trust math. Joe is logged into cs and wants to copy a file to math. On cs, he has set the environment variable *RNAMES* to be "jc@math". Previously, Joe edited his *.rhosts* file on math to contain "cool@cs". (At MCNC, for consistency, we have changed all relevant programs to use the 'user@host' format.) From the machine cs he types:

```
rcp file math:otherfile
```

Because Joe has set the *RNAMES* environment variable, this is equivalent to typing:

```
rcp file cool@math:otherfile
```

Since math trusts cs and since Joe has his *.rhosts* file on math set up correctly, the operation completes without asking for a password.

To see what is going on at a lower lever, consider the reverse situation. Joe is on math and wants to copy a file to cs. His *.rnames* file contains "cool@cs". He types:

```
rcp file cs:otherfile
```

Table 1

| Machine name | math | cs |
|---|---|---|
| Trusts (*/etc/rhosts.allow*) | cs | --- |
| User name | jc | cool |

*Rcp* calls *rcom* on the math machine. Since no remote user name is specified, *rcom* calls *rname* to obtain the correct remote login to use. *Rname* reports back "cool@cs". *Rcom* then calls *rcom_shell*, which talks directly to *rshd* on the cs machine. This remote *rshd* uses *ruserallow* to evaluate access authorization. Cs does not have math in its */etc/rhosts.allow*, so access is denied. The math *rcom* prompts Joe for his cs password, and calls *rcom_exec*. *Rcom_exec* talks directly to *rexecd* on the cs machine, passing it Joe's password, and establishes the connection. The same procedure would be followed if Joe had used *rsh* to cs. If he had wanted to *rlogin* to cs things would be slightly different. *Rlogin* calls *rname* to determine what remote name to use, then calls *rcmd*. *Rcmd* connects to *rlogind* on the remote machine which exec's */bin/login*. The remote */bin/login* calls *ruserallow* for authorization, which is denied. */bin/login* then prompts the Joe for his cs password and lets him in. Note that Joe's *.rhosts* file on cs does not come into play. Since cs does not allow password-free access from math, the *.rhosts* file on cs is never checked.

## 4. Conclusion

The MCNC networking changes are a natural extension to the Berkeley access scheme as represented by *rlogin*, but are far from optimal. A secure authorization scheme like Kerberos[2], for example, is a more correct approach. However, these extensions have been a useful intermediate step. They maintain a certain level of compatibility with sites which have not implemented them, while providing what we consider to be a reasonable level of security. As with all security measures, there are trade-offs. Here one trades off an increase in cleartext passwords passed around a monitorable network against otherwise unrestricted access. With more and more PC's and Suns allowing low-level network monitoring, the correct decision is less and less clear, but it is interesting to note that we have had widespread break-in problems in our area which spread rapidly through *.rhosts* access. During these break-ins MCNC remained largely protected because of our access restrictions. In light of this, we have made the correct trade off.

[2] Steiner, Jennifer G., Clifford Neuman, Jeffrey I. Schiller. "Kerberos: An Authentication Service for Open Network Systems," in *Usenix Conference Proceedings*, (Winter, 1988) 191-202.

# Modifying the Line Printer System for a Large Networked Environment

Elizabeth D. Zwicky
SRI International
zwicky@spam.istc.sri.com
Paul W. Placeway
BBN Systems and Technologies Corporation

July 28, 1989

At Ohio State, there are approximately 300 UNIX machines, which print to 16 printers. 3 of those printers are Apple LaserWriters attached to Appletalk networks; the remaining printers are attached 1 each to 2 Sun file servers, 2 each to 3 Sun diskless clients, 1 to a Pyramid, and 3 to a Hewlett Packard. Not only can any of these printers be reached by any of the UNIX machines that we administer, but the Appletalk LaserWriters are used from Macintoshes, two of the printers are used from a Dec-20, and one of the printers is used by a network of IBM RTs which we do not administer, and which does not have user IDs consistent with ours. In addition, some of these machines must continue to run vendor print spooler code, whereas we can modify the printing system at will for others. We have recently been able to consolidate our labs to the point where we are now confined to two buildings, although the printers are still relatively evenly distributed among four floors in one of the buildings. The result is a fascinating exercise in system administration.

It was clear to us that the vendor supplied printing systems would not fulfill all of our needs, but we could only replace the system on some machines, so the new system had to be fully upward and downward compatible with the vendor supplied code. Further, other parts of the printing system (notably **TranScript** and **CAP** depend on the way the standard BSD 4.2 system works internally.

There are several traditional ways to try to solve this problem, none of which we have found satisfactory. After many battles with the printing system, we decided that the easiest and most effective solution would be to simply fix the standard Berkeley line printer system, which already ran in some form on each of our machines. `lpq` was left largely unchanged, but we made changes to `lpr`, `lpd`, `lprm`, and `lpc`. These changes ranged from the trivial (for instance, making sure that all places where the programs exited with an error logged different error messages) to adding major features.

The features we added were aimed at improving handling of remote printing, improving control over individual jobs, and reducing the occurrence and severity of failures. They include:

1. Providing the ability to forward printer jobs from one printer to another from `lpc` via a `Forward` file in the spool directory, instead of by modifying the printcap file by hand. This file can be created by `lpc`.

2. Providing access to remote printers for hosts in a `hosts.lpd` file, as well as in `hosts.equiv`, so that hosts can have printer access without other sorts of access. This is a re-implementation of a Berkeley feature that would not work on all of our hosts.

3. Fixing `lprm` to be more flexible about host names and administratively equivalent hosts.

4. Improving failure detection and handling in `lpc` and `lpd`.

5. Improving administrative control over individual jobs.

We also are in the process of adding a number of other features to further improve the reliability and flexibility of the printing system:

1. Checking for and eliminating forwarding loops.

2. Adding code to automatically detect some types of failures, capable of automatically forwarding jobs to a backup printer.

3. Modifying the syntax of printcap to allow one printer name to refer to different printers in a context-dependent way.

## 1 Printer Forwarding

In an environment with multiple printers, it is relatively common for a single printer to be out of service, and for that printer's jobs to be transferred to another printer. The traditional way to handle this is to edit the `printcap` file on the machine with the broken printer. This procedure is prone to failure in multiple ways; in my experience, approximately half the times operators attempt to forward printing, the result is a damaged printcap file. This is despite the fact that we maintain our printcap files with commented out entries to be used for forwarding. Even when the forwarding itself works correctly, it is not uncommon for operators to inadvertently set up loops when two operators working separately forward printers to each other. We have even seen printers forwarded to themselves by accident.

Our solution was to have `lpd` check for a `Forward` file in the spool directory, after it has checked printcap. The `Forward` files contains a single line, consisting of "printername" or "printername@host". This file can be created by hand, or from lpc. The job is requeued for the specified printer.

## 2 Remote Printer Access

The line printer system normally checks hosts.equiv to determine whether or not a remote machine has authorization to use printers. In our case, there are hosts which we cannot put in hosts.equiv, because their numeric user IDs are not consistent with ours, but which have the same actual users. Since these hosts lack printers, we need to provide printer service to them. We therefore modified lpd to check not only hosts.equiv but also a new file called hosts.lpd. Hosts listed in either one are able to use the printers. We chose to make hosts.lpd used in addition to hosts.equiv, instead of in place of it, in order to simplify administration of our main machines; the idea of having to maintain another list of all 300 of our hosts was unappealing, and we do not have any reason to refuse printing to machines which are in hosts.equiv.

## 3 lprm Overzealousness

As distributed by Berkeley (and thus many vendors), lprm requires users to request removal of jobs from the same machine as they submitted them. There are two problems with this: the actual code was buggy and therefore sometimes failed to recognize a request that had been submitted from the original machine, and users fairly often forget where they submitted a job from. We modified lprm and lpd so that if the request for removal comes from the machine that submitted the job, or both the machine that requested the removal and the machine that submitted the job are in hosts.equiv, the job is removed. Root on the machine the job is currently spooled on may always remove jobs. All hostnames are resolved through the host database.

## 4 Single Printcap

There are two things that differ between printcaps in an environment where all machines can see all printers: the definitions for local printers, and the definition for default printers. For local printers, the same name will always refer to the same printer, but the definition for that printer will be different on the one machine that is actually driving the printer. For default printers (for instance "lp" or "postscript"), the same name will refer to different printers from different machines.

In the 4.3, Berkeley provided a simple change to the way printcap entries are parsed, which solves the problem of local printers; lpd is set to ignore rm: and rp: entries, if it is running on the machine named in the rp: entry, and to ignore everything else if it is not that machine. This solves two problems at once, because it also ensures that you cannot create a printer loop by creating an entry on machine foo for printer baz that says it is remote printer baz at remote machine foo; `lpd` will simply be unable to print those jobs.

The procedure for dealing with default printers is considerably more complex. It requires a conditional `rm:` entry, and that means modifying the structure of the printcap entry somewhat. We allow you to specify what printer name to resolve to by machine name or Yellow Pages (YP) netgroup, along with a default for machines not named. Machines that do not run YP can simply check against machine name, because machine names and names of netgroups are treated identically.

## 5 Failure Detection and Handling

Some types of failures happen with depressing regularity, including printer hardware failure, running out of paper without anyone noticing for an extended time (i.e. the middle of the night), or even someone simply leaving the printer off line after collecting a printout (common for line printers, less common for laser printers). All of these share a common symptom: the printer does not respond for an extended period of time.

In the new system, one can specify a set of printers to attempt to forward to, the initial timeout, and the time between checks after automatic forwarding has been started. If a printer does not respond for more than the initial timeout, the system goes through the list of printers to forward to, querying the server machine for each printer in turn. If that printer appears to be working, then the system starts forwarding to that printer. The system also sends out a message telling the operators on duty that it has started automatic forwarding.

After automatic forwarding has started, if the check time has passed when a job comes in, the system will try to handle it locally again, in the hope that the printer has been fixed. If it has not, forwarding is continued. Forwarding may also be turned off by hand.

The `status` command of `lpc` was also expanded to check for a number of different software and human failures that the printing system suffers frequently. These include checking for correct ownership of the spooling directories, better checking of the syntax of the printcap file, and recursively asking for a remote status for non-local printers.

`lpd` attempts to detect printer loops, another common cause of mysterious failure in our environment. To preserve compatibility with normal printing systems, it does this by adding a small data file containing a list of the past printer and host combinations the job has been queued for. This data file is deleted after the job has printed. If `lpd` finds that a job is queued for a printer and host combination already listed, that queue is suspended and the status for that printer is set to "Printer is part of a forwarding loop."

As a final aid to detecting errors, we standardized the use of `syslog` to record errors produced by `lpd`. In order to avoid overfilling syslog files, messages produced when debugging is turned on are logged to the file named in `lf`.

## 6 Control Over Jobs

The line printer system normally provides the ability to remove jobs, move them to the top of the queue, and reject them on the basis of their size in bytes. Unfortunately, none of these facilities is quite what we usually need.

`lprm` removes jobs by job number, and it is not at all uncommon for several jobs in the queue to have the same job number in a networked environment. Job numbers are assigned on the machine that originates the job, and do not change as jobs move between machines. If you recreate clients from a single master, as we do, every time you do a mass update you resynchronize the job numbers on all clients. Printer queues immediately after such an update often have 20 jobs, all with the same job number. To resolve this ambiguity, users may now specify "jobnumber@host".

We occasionally need to move critical jobs to the top of the queue, but we more often need to move large jobs to the bottom of the queue. `lpc` did not originally provide this facility. We added an `lpc` command "bottomq" (due in concept to Bob Manson), to move specified jobs to the bottom of the print queue (the inverse of the existing "topq"). Again, both topq and `bottomq` require the ability to specify job number and host, not just job number.

Filtering jobs by size in bytes is reasonably accurate when you are using a line printer. Unfortunately, size in bytes is not well correlated with size in pages when you are printing to a PostScript laser printer. Size in pages may or may not be determinable, depending on the type of file. Our solution was to use a filter which knows about the most common types of files we print (plain text, Adobe standard PostScript, and the PostScript that `dvi2ps` produces), and to treat the remaining files as OK to print. The maximum number of pages is specified with a `mp#` entry in printcap, and defaults to infinity.

# Administration of a Dynamic Heterogenous Network

*Richard W. Kint*
*Charles V. Gale*
*Andrew B. Liwen*

Software Productivity Consortium, Inc.
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070
{kint|gale|liwen}@software.org
+1 703 742 8877

## INTRODUCTION

### History

The Software Productivity Consortium was founded in 1985 by fourteen aerospace companies with the objective of developing products and methods to facilitate building large mission-critical systems. The Consortium opened its doors in the summer of 1986, with an initial computing plant consisting of a Gould UNIX supermini with 10 terminals and a Wang word processing system. Workstation installation began January 1987, and the network had 100 workstations by late March of that year. There are currently around 200 systems at the Consortium running on a building internet.

The vast majority of the work done on the Consortium network is software engineering or related activities (documentation, USENET news, *xmazewar,* etc).

There are three distinguishing attributes about the Consortium network. First, it started from a clean slate. The Consortium was a startup, with no corporate precedents to influence decisions; this proved a mixed blessing. Second, it grew very quickly – the network had over 100 systems within three months. Third, as a medium-sized network in one building, it is centrally administered – users have no special control over the node on their desks. Since configuration consistency is one of our main goals, this means that many actions must be performed on 150 systems concurrently. This requires developing methods and programs which automate routine jobs and which guarantee that operations are applied to all relevant systems in a timely fashion.

This paper will present a description of several interesting aspects of the Consortium network and the way it is administered, and then take a more detailed look at several problems and our solutions to them. We hope that our solutions will help others bypass a painful learning process.

### Site Description

The Consortium uses networked workstations nearly exclusively, the exceptions being a VAX 8550 (at the high end) and PCs and Macintoshes (at the low end). Until recently there were a couple of UNIX superminis, but these proved less cost-effective than workstations for our purposes and were phased out. There are four classes of platforms:

- Apollos (c. 150 user nodes, c. 20 other)
- Unix Systems (Suns, c. 20)
- VAX/VMS Systems (c. 20)
- Miscellaneous (Symbolics, TI Explorers, PCs, Macs)

The view of the network and of resources is consistent within each of the first three classes of platforms and "sort of" consistent between Apollos and UNIX systems. There is little consistency between other classes.

Although the only native UNIX systems we have at present are Suns, past experience has shown that BSD-based systems supporting NFS can largely be administered as a group. Apollos run a proprietary operating system which offers UNIX environments, but for administrative purposes they differ sufficiently from native UNIX systems that they are handled as a separate class.

For the most part, the VMS and miscellaneous systems are autonomous fiefdoms within the central support organization. We intend to change this over time but have only begun the process. Accordingly, the VMS and miscellaneous systems will only be mentioned tangentially in this paper.

### Workstation Distribution

Most technical personnel have either an Apollo or a Sun on their desk, with some having a VAXstation or a Macintosh as well. A couple of Macaholics have only a Macintosh, but they seem content and it frees up a workstation for someone else, so we don't disturb them. Non-technical personnel have either an Apollo or a PC.

### Network Disk Configuration

Most Apollo and Sun workstations have local disks, which total some 32 GB. There are a few diskless systems, but these have proven troublesome. Central mass disk storage is provided by Apollo disk servers (2.6 GB) and a Sun Data Center (1.5 GB). The VAXstations use local disk primarily for paging, with disk space provided by a MicroVAX. Backups on the Apollos and Suns are done using 8mm tape drives, which store 2.56GB of data on a cartridge costing $7.00. A high-density backup medium is required for large amounts of disk.

### Communications Backbones

There are two physical backbones: an Ethernet (four physical networks, one logical network) and an Apollo Domain token ring (four physical and logical networks). The token rings run the Apollo token ring protocol (with TCP/IP encapsulated therein), and the Ethernet is used for TCP/IP, DECnet, and Chaosnet traffic. There are probably a couple of small Appletalk networks that the exterminators missed.

the whole, we have found the token ring preferable to Ethernet: it's either up or it's down. When it's down, ing the problem is generally quick. Ethernet has too many degradation modes for comfort.

### Connectivity

Nearly every system can communicate with every other, with TCP/IP often serving as the lowest common denominator. Apollos speak Domain and TCP/IP, Suns speak TCP/IP, Vaxen speak DECnet and TCP/IP, and the LISP workstations speak Chaosnet and TCP/IP. Some PCs are hooked into the Apollo network through serial ports (providing only a remote login capability) and some Macs speak TCP/IP. We still find ourselves using SneakerNet from time to time, and once had to resort to dialing in to get a Mac on the network.

### File Sharing

Apollos can transparently share files through Apollo's built-in network file system, can share files with Suns through NFS, and can share VAX files with an Apollo product. Suns can share files with other Suns and with Apollos through NFS. The VMS systems are mostly clustered together (they will all be clustered soon), but do not share files with other systems. We plan to install NFS on the VMS systems soon, so they will become full file-sharing network citizens.

### Brownian Node Motion

The Consortium is a dynamic organization — people come, people go, people move, divisions get reorganized, equipment comes, and equipment goes. All of this involves a lot of equipment moves. This has turned out to be one of the major disadvantages of workstations. When a workstation arrives, leaves, or moves, there is quite a bit of overhead. Databases and host tables have to be updated, accounts need to be changed, etc. As a general policy, we try to wipe and reload nodes assigned to new people. Contrast this to the work required to move a terminal. This cost was not anticipated during the network design.

### User Accounts

Since NFS is used between Apollos and UNIX systems, UNIX uids need to be consistent across these systems. Since Apollo uses a different authentication scheme internally with UNIX uids being a mapping onto this, all accounts (and thus uids) are created on the Apollos and the uids exported to the Suns when setting up accounts.

This was not done originally; initially the UNIX systems and the Apollos had separate uid spaces. When NFS was installed on the Apollos, merging uids was a massively painful experience. VMS UICs are completely separate and we anticipate loads of fun when we install NFS on the VAXen.

**Local Admin Software**

Local administrative software consists of scripts (written in the native command interpreter or in *perl*) and programs (written in C, using the UNIX interface insofar as possible). As long as the more obscure features of UNIX (such as `/dev/kmem`) are avoided, UNIX code is easily portable to Apollos and code which does not involve process creation or signal handling is surprisingly portable to VAX/VMS.

**Site Licenses**

We have found node-locked software to be essentially unusable. The Consortium obtains site licenses for most software used on workstations. As more innovative software licensing schemes become standard (such as network license servers) we hope to be able to use them.

**Unified Network View**

The Consortium divides stages in the software lifecycle among three divisions: one evaluates ideas for tools and prototypes them, one develops tools based on those prototypes, and a third division tests the tools..

This has implications for the design and policies of the network — since the divisions are organized around particular lifecycle phases rather than projects, a tool (or prototype) is passed from one division to another at least twice. This means that software which works on one node had better work on a node in another division, which necessitates a uniform network with a universal view of the filesystem.

**Centralized Administration**

The Consortium takes the view that a workstation sitting on a person's desk does not belong to that person, but that it is a piece of the corporate computing system. In a network, decisions taken on one host can have unintended but wide-ranging effects.

The proper business of software engineers is software engineering, not system administration. Accordingly, all administration is handled by a central administrative organization. Users do not have the root password (or equivalent) for the machine on their desk. This has the advantages of consistency (all nodes receive the same level of support), uniformity (all nodes look the same), and thoroughness.

The primary disadvantage of centralized (or fascist) administration is that in emergency situations it takes time for central administrators to get to everyone's system; for those users capable of administering their node, this seems an unneeded delay. However, the set of users actually capable of administering their nodes in a network is a subset of those who think they are capable of administering their own nodes so we see no change in this policy.

**Neutral System Names**

We started out naming workstations after the user in whose office they sat. This seemed like a good idea at the time in that the node name was quite meaningful, but the Brownian Node Motion mentioned above made this unusable (for some reason, users didn't want hosts with someone else's name). In addition, some names were difficult to type without making mistakes. We now give all nodes short neutral names. These names are given to nodes on arrival, even before they are placed in offices. We toyed with letting people name the node in their office, but this proved unworkable.

## PROBLEMS AND SOLUTIONS

### NODE INSTALLATION

Initial system installation is time consuming and cumbersome; it requires too much repetitive keystroking, with the concomitant chance of error. Since several products from several vendors form the basic node configuration, and since we modified software configurations to meet local needs, there was also cleanup left to be done after all software was initially installed.

We tried copying software from active systems, but found that this does not work well in practice. There is too much chance of something being changed inadvertently, which led to a problem of bad software being propagated throughout the network.

### Node Templates

To cut down on this burden, we built a "perfect node" user node template which contains all of the standard software and is copied onto all user nodes being installed. Permissions on files are set and checked manually. In addition, the installation script creates all system-specific configuration files. This guarantees that all nodes start from a consistent base.

The template configuration is tracked using *DSEE*, Apollo's version control system. As changes are made to the template, they are logged so that old configurations can be recreated. One weakness of the system is that this is enforced only by convention which may be bypassed; on occasion changes were made to nodes in the field but not to the template, creating inconsistencies when nodes were installed from the (outdated) templates.

Creating the node template involved installing some packages using the vendor-supplied methods and reading some things off tapes ourselves, bypassing the installation methods. One major advantage of script-based installation systems is that they facilitate "roll-your-own" installation methods — all the necessary information is in the scripts, even if tracking it down is cumbersome. Binary installation programs hide this information.

### Cost

The tradeoff involved in the node template is between disk space and effort (template creation and maintenance) on the one hand, and the time saved doing installations on the other. A user node template is 25-30 MB, and we maintain four of them (three for different O/S versions, and one for a special node configuration). Template creation takes the same amount of time as loading a node by hand, typically two to three hours (including chasing down problems). Maintenance is not time-consuming, but ensuring that all changes made to nodes in the field are applied to the templates has proven difficult. This is purely a procedural problem.

### Savings

rimary time saving is in the "load and go" nature of the installation process. Nobody need monitor the instal- n to make the necessary responses, or change tapes. Once started it can be ignored until someone comes back to check for errors. This time savings amounts to at least one hour per system installed.

There is also a considerable savings in time and aggravation realized from having all the loose ends taken care of up front. When installation is done manually, it's easy to forget a detail which must then be tracked down later. This is not a problem when templates are used. We have also eliminated the chance of someone typing in the wrong response in interactive installations.

By using these templates, six people were able to update 150 user node Apollos to a new version of the operating system over one weekend last year. We found that the primary bottleneck was the disk containing the template, not network bandwidth, so we copied the template to several nodes and split the network into several pieces.

The template system has also helped our testing operation. The Consortium tests software on "clean nodes," systems with only a minimal set of software installed and no local modifications. This guarantees that Consortium software will work in other environments. A clean node template is maintained as well, so periodically the clean nodes can be wiped and reloaded from scratch. Since this is now a painless operations, creeping nastiness is avoided at a very low cost.

Since the overhead is high, templates are only used for Apollo user nodes. As we get more Suns we anticipate doing the same thing with them. Since Apollos do not require separate swap space, there is little gain from partitioning the disk on a workstation and we do not partition disks on the Apollos. Since Suns require at least three partitions (including swap), this will complicate the templates slightly but does not seem a large hurdle.

## Software Configuration

The Consortium has a very rich software environment; some 65 software products are available (in addition to the operating system utilities) for the Apollos, and some 45 for the Suns. This amount of software will not fit on a workstation disk. Local disk space is comparatively expensive, while central disk space is relatively cheap. In addition, we needed to keep the consistent view of the network — all software should be equally visible to all users, with exceptions such as node-locked software. So we had to divide up software into local and remote classes. Local software is installed on the local disk, while remote software lives on another disk and is accessed through remote mounts and links.

The scheme below applies to the Apollo systems; a similar setup is used on the Suns but is less detailed due to the smaller number of systems.

We wound up with four levels of software:

- Software installed on all nodes (**local** software)
- Software installed on site nodes (**site** software)
- Software installed on a central node (**central** software)
- Software installed as needed (**special-case** software)

In the first case, software is installed everywhere (150 to 170 nodes). At the other extreme, in the third case software is only installed in one location with other nodes linking to it. The fourth is the catchall for special cases; it refers to software installed on some nodes but not all. The second case (site software) is a middle ground. A site node is a file server/administrative node with 700 MB of disk space which provides replication of resources for fault tolerance and load distribution. There are five site nodes on the Apollo internet. Site software is software which is installed on all of the sites, so that it is centralized but also replicated.

**Criteria**

The criteria used in deciding how to install a given software package belonged to are:

Necessity

As software becomes more critical, it should be more distributed. Software that is needed even when the network is down must be local. Local software always works when a network is down unless it accesses external resources. Central software is unavailable if a network dies, and is usually unavailable in the event of gateway failures. Site software is available if the local network is up.

Frequency of Use

As software is used more, it should become more distributed to increase fault-tolerance and to spread the load among disk arms. Less-used software should be centralized. All commonly used packages are either local or site software.

Behavior

As software causes more network activity, it should become more distributed. Local software causes little network paging, while central software access generally goes through gateways. Site software again offers a middle ground – network activity is limited to one network.

Size

Larger software packages should become more centralized due to the cost of local disk.

Ease of Administration

Software requiring frequent maintenance will be more centralized; it is much easier to maintain five copies of a package than 150.

Permanence

Installation and administration is expensive, so software that is installed for evaluation or testing is installed as central software or special-case software.

**Overlapping Criteria**

As can be seen, the criteria above can be contradictory. For example, we use Interleaf extensively. It's big (15 MB), which argues for centralization, but it is used all the time, is necessary, and can cause lots of network paging, all of which argue for localization. So we did both.

At TPS version 3.0, the main Interleaf directory had five main subdirectories. Each was moved off-node in turn and the network activity (and the subjective delay) were compared. Those portions that caused a lot of network activity or seemed to delay execution were made local, and the remainder placed on the site. In this way, we split a 15 MB package into a 2 MB local portion and a 13 MB remote portion.

The directory is the customary level of granularity for splitting packages. We have been more specific in rare cases, but it's generally too much hassle to deal with individual files.

**Summary**

To summarize, base operating system executables and the most-used tools are installed on all nodes. Most other software, including operating system text files such as man pages, was installed on site nodes. Software that is only needed occasionally or by a small number of people is installed on a central node. Special-case software (such as software required by a special hardware device, or node-locked software) is only installed as needed.

## SOFTWARE DISTRIBUTION

Templates take care of initial node configuration, but updates also required attention. A set of simple, but effective, tools were developed to handle software distribution. These tools are simply front ends to standard system commands such as *cp*, *rsh*, and *ln*. The enhanced commands look exactly like their normal system counterpart syntactically but are able to iterate the command across a set of hosts. Each tool normally has the same name as the system counterpart but with a *"g"* (for global) as the first character. The idea for this kind of tool first came from scripts included in Larry Wall's *perl* program distributed on Usenet and, in fact, these tools have all been built using *perl*.

**Comparison to Other Methods**

This approach was used over *rdist*, Apollo's *RAI*, and subscription oriented distribution tools for several reasons. First, these programs were not available on all platforms. *Rdist* is not available on Apollos and *RAI* is not available on the Suns. Second, neither program is flexible enough for such a dynamic environment. The centralized node administration, the large number of software packages, and bizarre licensing schemes currently prevent the use of subscription based systems such as *SPUDS* and *Track II*.

This approach to building front ends to system commands has several benefits. Administrators do not need to be trained how to use these tools since they should already know the existing system counterparts. Software distribution across the network can be accurate without each administrator having to know which nodes have been added to or removed from the network. The administrator can control exactly which nodes receive full software packages and which nodes only have links to site nodes or central nodes, as described in the previous section on Software Configuration.

**Implementation**

The host set names are contained in */etc/ghosts*. This file is created automatically by *cron* each night to make sure it is up to date. The file is created by a *perl* script from hardware information kept in a relational database and by information gleamed from the node filesystem. The relational database is kept up to date by technicians as they change hardware. Examples of some of the host sets used are: *APOLLO* (all Apollos), *SUN4* (all Sun 4s), *RING1* (all Apollos on the first token ring), and *DEE* (all machines used in the DEE division of SPC). The sets we commonly use are based on the machine type (where there are differences), the network the node is sitting on, the site node this node is linked to, and the company division that owns the node.

An administrator may also keep a personal *ghosts* file to contain set names needed individually. For instance, one *ghosts* file contains a list of hosts in the set *DEE_ADA* for all the DEE division nodes licensed to use a certain Ada compiler. All the tools look in the current directory for a *ghosts* file and read it if it is there; then they read */etc/ghosts*.

The scripts detect any errors when trying to execute a command on a remote node and will add the host to a special set called *ERROR*. This allows the administrator to ignore these hosts in following commands, or, after correcting the problem, to re-execute the same command again on just those hosts.

**Example**

Here is a small example of how one of the commands operates. The *rcp* command has an enhanced counterpart *gcp* (global copy). *Gcp* is exactly the same as *rcp* syntactically, but the hostname given on the command line is expanded into a set of hosts and the *rcp* command is iterated across each host in the set. For instance, to copy */etc/passwd* from your local host to a remote host, the command is:

```
rcp /etc/passwd node:/etc/passwd
```

To do the same task to a group of hosts, the command is:

```
gcp /etc/passwd hostset:/etc/passwd
```

This will copy */etc/passwd* to every host in hostset.

**Problems**

Uniformity across all platforms is still a problem with this approach. Most of the commands built on top of standard Unix commands work on the Suns and Apollos. The exceptions are commands such as *df* which rely on network file system naming conventions (Apollos use *//nodename*, NFS mounts remote systems below root). Commands built on Aegis commands only work across the Apollos and none of the commands work for VMS. We are looking at solutions to this problem, including a port of *perl* to VMS.

## Configuration Control

Maintaining configuration consistency has proven to be a continuing problem. The ease of use of the *g*–commands (see *Software Distribution* above) has actually exacerbated the situation; when updating nodes was more cumbersome, the overhead of keeping the template current did not seem as comparatively burdensome. The ease of use of the *g*-commands has made *ad hoc* configuration changes easier. In addition, sometimes benign changes have malign side–effects which are not realized at the time.

To meet this problem, we have developed a set of tools built around a manifest, a file containing file names and a set of attributes which looks like an *ls –l* listing,. By generating a manifest for a template and then comparing each node to this manifest, deviations can be monitored. This has not been fully integrated yet; two problems are sorting out insignificant changes (such as */etc/wtmp*) from significant ones, and separating out known differences without having to wade through them each time.

## User Support Dynamics

While the Consortium approach to system administration follows a centralized model, user access to the support organization employs a totally decentralized tack. Early attempts at servicing the user community resulted in many frustrated attempts at "problem desk" tracking schemes. User's could call in or stop by the support group to report problems or request support services. Our experiences with this model proved unsuccessful, especially as the size of the user community grew.

As we attempted to make the system work, some of the problems we encountered were:

- User requests inevitably got lost. Anyone who has tried to track problems in even a moderately sized network has already realized this fact.
- Requests for "simple" services, such as copies of documentation, were delayed because it meant transferring the request to other service providers. Often the service provider was free and could provide the support right away.
- Sometimes requests were delayed because it meant physically carrying a request form to another desk or office in the facility.
- Frequently problem resolution required lead time. For example, a manual might need to be ordered or a unit sent out for repair. Even though adequate attention had been given the problem, the submitter was not aware of this and his frustration grew.
- Because of time involved in reporting corrective actions, a problem might be fixed, but there could be a delay before the requestor realized it.

While the Consortium was small and growing (up until mid–1987), requests could be tracked in an *ad hoc* fashion. However, as the organization, grew this broke down, which led to a loss of confidence from other elements in the organization even though the support group was working harder than ever.

**Approach**

The approach had to be a model that used effective, automated tools and that included the users in the process. The support organization received a charter to implement a number of user friendly, graphical tools for this purpose. Two of these tools are discussed briefly below.

**The *Request Tracking System***

We created a proactive system where the user could initiate requests for any type of service. The system was built around an Apollo Dialog front end that combines a point and press metaphor for action selection. The interface style is built so that users can figure out what to do with almost no instruction.

An important aspect of this system is that users do not have to determine ahead of time what service component in the Consortium can handle their request. Everything from "My node won't boot" to "I need floppy disks" can be described. The ease with which problems are reported has meant that users are more prone to report all problems through this system.

Problem Entry

A friendly (albeit somewhat busy) panel greets the user when *RTS* is invoked. Once the ***add*** operation is selected, a "forms" section of the display is energized. A one line problem description is entered and then an editor pad pops up into which an extended description may be entered. Thereafter, the user can assign relative priority to the problem. At that point a record is entered into the data base and a problem number assigned.

Once the problem is entered by the user, he has the opportunity to modify any portion of his submission. Only after selecting the ***confirm*** button will the request actually be submitted. The user may opt to cancel the transaction at any point beforehand.

Problem Assignment

When entry is confirmed, the user portion of the *RTS* script then sends electronic mail to Network Services, a generic user on the network. NS maintains a problem desk which monitors new mail and analyzes problem submissions.

User Feedback

Once a request is received by NS, confirming e-mail is sent back to the requestor. This important component gives rapid, direct feedback to the user that his request has been seen and is being worked on. NS can annotate the reply message to inform the user if a particular request will be delayed or if other exceptions will occur.

Once a request has been handled, the RTS system also sends mail notifying the user of completion. This is accomplished automatically even if the request is suspended for whatever reason. During the time that the problem is being repaired, the user may invoke the RTS tool and query the status of his request. This process provides complete life cycle feedback to the user during the entire tracking process.

**The *swupdate* tool**

The Consortium archives a very large number of pieces of software. There were 5 platforms (Apollo, VAX, Sun, Gould, and TI) to deal with. These platforms, of course, had their attendant OS revisions to track. In addition there were a number of system utilities with their own independent release cycles which required tracking. These include, for example, NFS, NCS (Apollo's Network Computing System), and various flavors of print spooler drivers.

In addition to the operating system were the myriad number of "standard" applications. For example, our common desktop publishing tool is Interleaf. Each platform had to track its own release cycle of Interleaf. Because some platforms could be using older versions of the application, interchangeability of data among the tools and platforms has to be managed as well. Other common tools found on all Consortium platforms were the data base tools, OTS CASE tools, and spread sheets. Each tool exponentially increased tracking problems when combined with other platforms and with other tools.

Certain Consortium tools harness third party tools. Because of this, there is a constant stream of software which must be evaluated for "harness-ability" with Consortium tools. This type of software lives on the network under varying terms — sometimes only a small set of nodes is licensed for evaluation, and sometimes all nodes are licensed for use but the duration of the license is foreshortened.

Initial Approach

The centralized administration model used on our network proved especially necessary for software administration. Each software package often had distinct licensing requirements. We had to mandate that all software on the network flow through one point in order to exercise the necessary accountability, so we created a role of *software administrator* within our support organization. This policy at least created a control point though it didn't necessarily make the software administration role any easier. All users were forced to deal with the control point for almost all software issues.

Initial attempts at managing the software base involved an *ad hoc* data base which tracked packages, versions, and nodes on which installed. This simple data base was useful to the software administrator but proved less useful to others searching for information on specific software.

Initial Tool

Users often need to determine what software was installed on which machine. They had to evaluate it, or track changes in a recent version, or help steer changes in a future version. Again, the obvious solution was to provide a tool for them.

The initial product presented a flat hierarchy of all topics tracked by the software administrator. Queries could be restricted to recent installations and beta software. This tool proved very useful during the first phase of software administration. Unfortunately, maintenance of the data base remained a manual process. Since everything flowed through one office this was workable – until the operation grew.

Soon the job was too large and installation tasks had to be delegated to other members of the administration group. The office of software administrator retained sole authority to update the data base. Since several others were performing software updates, these were not always entered in the data base in a timely fashion.

Current Tool

Recently the *swupdate* tool has been significantly revamped. Data base functions have been modified to allow anyone making an installation to update the data base. Because update is easy, information can be entered into the data base before the software actually arrives for installation. This allows much more thorough life cycle tracking.

A completely new user front end has been added. The user now sees a smaller topic set with a richer modifier base. This allows more specific browsing by encouraging more targeted selection of topic area. Additionally, cross-topic or multi-topic browsing is more easily accomplished.

### Summary

We found involving the user with the reporting and management of his own requests extremely important in a large, diverse network. It was especially important to provide easy to use, high quality applications to allow the user to monitor the progress of his own requests. Devoting the time and resources to providing applications which support this model is critical, once a network size reaches critical mass. Our experiences show that a small number of machines can be managed for a while without these tools, but once things get bad they get bad quickly.

The primary disadvantage of the *RTS* tool is that the Problem Assignment phase introduces a time lag which can be upsetting to users who require immediate action. The volume of requests is such that without this buffer nobody in the support group would ever get anything else done, but it would be nice to meet both needs.

## CONCLUSIONS

Running a heterogenous workstation network is a complicated and expensive process. The investment in tools to automate the administrative process will pay off over the long term, even though the commitment in manpower can seem frightening,. It is best to recognize problems up front and commit to solving them. Ignoring them will only chip away at overall long-term productivity.

# Implementing a Consistent System over Many Hosts

*Nathan H. Hillery*

Computer Science Laboratory
Department of Computer Science
Duke University
Durham, NC 27706

## Introduction

Over the past five years, the Department of Computer Science at Duke University has experienced an amazing explosion and transformation of computing resources. Five years ago, the computer system was one PDP-11/70 tied to 2 PDP-11/34's and one Vax-11/750 via serial lines. Most of the load was on one central machine and it suffered due to inadequate resources. The PDP's were supplanted in 1985 with 3 Vax's with ethernet interfaces, but still most of the load was placed on one machine. The satellite machines were used to support research groups and each was a unique system. In this same time frame, the department's first workstations were installed in six graduate student offices, along with a server that became the de facto central machine for graduate students. Later, when faculty members started getting workstations of their own, another server became the faculty central machine. By the end of 1987, there were twenty diskless workstations and four servers, with an expected doubling within the year. Needless to say, as the number of workstations and servers increased, there soon would be too many "central" machines! Likewise, the load on our network was already high and would soon be unbearable. These factors forced the systems administration staff to reexamine the architecture of the system and radically change it. From this reorganization came some valuable new lessons and reaffirmations of old ones.

## Site Description

The Computer Science Department maintains modern computing facilities to meet a variety of research and graduate teaching needs. The list of larger computers includes:

- a Convex C1 (96 Mbytes memory, .5 Gbytes disk),
- a second Convex C1 (32 Mbytes memory, 1.5 Gbytes disk, 1 tape drive),
- a BBN Butterfly GP1000 (64 nodes [each with 68020, FPU, 4 Mbytes memory], .5 Gbytes disk),
- five Sun 4/280's (each with 32 Mbytes memory, 1.8 Gbytes disk, 1 tape drive),
- a Sun 3/280 (32 Mbytes memory, 1.2 Gbytes disk, 1 tape drive),
- two Sun 3/180's (each with 8 Mbytes memory, 1.2 Gbytes disk, 1 tape drive),
- and a Sun 3/160 (8 Mbytes memory, .8 Gbytes disk).

These computers have a total of 608 megabytes of main memory and a total of 15.9 gigabytes of disk storage. These all use versions of the UNIX operating system (SunOS for the Suns, Mach for the Butterfly, and Berkeley 4.3 for the Convex's). The Convex is a vector processor suitable for numerical analysis and simulation, while the Butterfly is a parallel processor supporting research in parallel data structures and algorithms. The other computers listed are for general research and education purposes.

Workstation-sized computers in the department include:

- twelve Sun 4/60C-8 diskless color workstations,
- ten Sun 3/110 diskless color workstations,
- twenty-two Sun 3/60 diskless monochrome workstations,
- twelve Sun 3/50 diskless monochrome workstations,
- a Symbolics 3670 lisp machine,
- five AT&T 3B2/310 microcomputers,
- and five AT&T 3B2/400 microcomputers.

Most of these, again, use some version of the UNIX operating system (System V for the AT&T machines, various implementations of Berkeley 4.3 for the others) while the Symbolics offers a unique LISP-based environment.

A Public Computing Laboratory was established cooperatively by the Computer Science Department and the Center for Academic Computing in Fall 1988. It includes two clusters, each with a Sun 3/180 file/compute server with 1.8 Gbytes of disk and 10 Sun 3/50 workstations. This gives students and faculty exposure to a UNIX workstation-based environment.

Workstations and terminals are installed in each student office. Each staff office is equipped with a terminal or workstation as well. Public workrooms provide access to more workstations and terminals. The department also has several IBM PC AT's and many IBM PC's. Macintosh's are installed in some offices and in a public workroom. Peripheral equipment includes four Imagen 8/300 laser printers, five Apple LaserWriters, and two Hewlett-Packard color plotters.

There are a variety of supporting communications facilities including Ethernet connections for the larger computers and the workstations, asynchronous communications for all terminals and computer ports through an AT&T ISN dataswitch. For off-site users, there are 12 phone lines dedicated to incoming modems and 4 for outgoing modems. Both 1200bps and 2400bps service is supported. Remote concentrators attached to the ISN through fiber-optic cables extend communications to the School of Engineering and the Department of Physiology and their respective computing resources. Microwave links provide high-speed data paths to computing facilities to several other sites in the state of North Carolina. Our machines are connected via these links to NSFnet and the rest of the Internet.

Facilities on campus maintained by the Duke University Computation Center include an IBM 4381 and approximately 140 personal computers to which the department has priority access for two labs containing a total of 70 IBM PC's.

Further computing power is provided to the department (predominantly for VLSI design work) by the Microelectronics Center of North Carolina (MCNC). They run two DEC VAX-8600's, two VAX 11/750's, and two CONVEX C-1 systems. These systems all run Berkeley 4.3 UNIX and are connected to the Department of Computer Science via the MCNC Microwave System. MCNC serves as the central site for a state-wide communications network and as the gateway to the Internet.

The North Carolina Supercomputer Center (NCSC), housing a Cray Y-MP4, is to be opened in Fall 1989. A high-speed connection will be established between NSCS and the deparment providing workstation users supercomputer access.

## Guidelines

The turning point in the direction of computing at Duke CS was motivated by one central prediction - that soon everyone was going to have a workstation on their desk. With this as a starting premise, we fairly quickly developed some guidelines for how we wanted to direct this migration:

- All workstations would be diskless (we have recently relaxed this to allow our first dataless client).

- Compute cycles on all general-purpose machines would be treated equally, to achieve load balancing (this is even more important during the transition phase before some have workstations).

- The network would be reconfigured to support increased traffic levels and allow unhindered growth of scale (but should still appear as a single network to users).

The first guideline stems from a desire on the systems administration staff to retain central administration and maintenance even though the computers were being distributed. Disks require added attention that is easier to provide in a machine room than in someone's office. The performance tradeoff, in our experience, often favors going through the network to access a large, fast disk on a server more than going locally to a slow, small disk. Dataless clients (workstations with a disk that holds only root and swap) still carry added maintenance concerns, but don't require the degree of administrative attention needed for diskfull ones.

Diskless machines also make it much easier to move users. Our department in arranged so that students may move from one office to another (potentially on a different floor or wing) each year. To accomodate this and still keep inter-machine dependencies to a minimum, we move a student's files when they physically move. Although this requires some work on our part, it is more that made up by the consistency this offers. Simply by knowing which office a student is in, we know which server holds their home directory. Not leastly, lowered inter-dependencies appears to keep more users happier. Planning for future growth is facilitated as well, since we know approximately how many faculty and students can occupy a range of office. The connection between a server and the clients it serves is flexible enough to allow for unexpected turns of events. We can now make well-grounded statements about how facilities will have to be reconfigured in order to support additional clients (even to the "one workstation to one person" level). When workstation failures occur, swapping out a diskless workstation is trivial mainly because no data has to be copied and little changed.

Treating cycles on all machines equally basically means that everybody can login to every machine and most operations work the same everywhere. The main criterion a user should use when selecting a machine to work on is how well it gets their job done. If all machines can do most things, then users following their natural instincts should bring about a balancing of load. Allowing this balancing may slow down the most intensive of activities but it allows the bulk of computing to be done more efficiently. The most obvious action the lab staff took to promote load balancing was giving most users an account that permitted access to almost all machines in the department. Each user has one, and (with a small number of exceptions) only one, home directory that appears in the same place in the directory tree on all systems. Another significant factor was that just about every non-system filesystem is NFS mounted on all systems, making all systems look almost identical. Except for certain special-purpose (or expensive) tools, software is installed identically across all servers. For terminal and modem traffic, we have installed two home-grown Ethernet terminal servers that allow selection of host groups in addition to host names. These host groups are comprised of machines intended to support specific kinds of users or activities. Selecting host group "grad", for instance, connects the user to the least loaded of the workstations installed in grad student office. The naming scheme of the graduate student workstations (grad[1-21]) is another indication of how generically they tend to be treated.

As we saw our network grow, both in the number of connected machines and amount of traffic generated per machine, we became convinced that the network could not remain intact for much longer. When the collision lights on our multiport repeaters started being on almost continuously, we knew we would have to do something right away. The choice was made to divide the single network into multiple subnets tied together through a backbone network. We also chose to integrate the multiple networks in a manner that would make gateways transparent to our users. We hoped that this could be done in such a way to keep problems somewhat restricted to the subnet on which they arose.

## Subnetting Solutions

It became clear as workstations are added to the system that the network was becoming overloaded. Several schemes for subdividing the network were investigated including routers and gateways. We chose to subnet with gateways for two principle reasons. At the time of the decision (late 1988), the cost ratio between routers and gateways (for us, an additional Ethernet interface) was about 8:1. It was felt that the network was strained so badly and the number of future additions to the network so high that only the addition of three subnets would be sufficient to help (by the end, we had added four). We could not overlook a difference of approximately $22,000. The second factor in our decision was our recognition that by making gateways be servers and their subnets solely their clients we could subnet without decreasing reliability. Users had already learned to accept dependency between their workstation and their server. While the server was down, their workstation would be hung, but would continue after the server came back up. With this scheme of subnetting, if the server is down then none of the diskless machines can function so the users don't care that they can not communicate out of their subnet. Contrarily, with a router, if the router is down the clients are kept isolated regardless of the state of their server.

Keeping all diskfull machines on the backbone network made it so that all NFS activity would never have to go through more than one gateway (and that most would go through none). A workstation user accessing files from a server other than the one serving the workstation necessarily has a dependence on two machines. Subnets, as we implement them, do not introduce any additional dependencies. Since NFS file access and swapping is often the bottleneck, arranging the network so that NFS traffic through gateways is minimized maintains good workstation performance.

## Routing Concerns

The Computer Science Department network is connected to a large Class B network (tucc-mcnc, 128.109) that is divided into many subnets. A gateway (suranet-gw) ties the tucc-mcnc network to SURAnet and NSFnet. Hosts on the backbone use the standard Class B netmask (0xffff0000) while all the gateway hosts use a subnet netmask (0xffffff00). At boot time all machines on the backbone network establish a default route through suranet-gw. This requires a slight trick on the gateways since with a subnet netmask they will not be able to reach suranet-gw because it will appear that it is on a different network. The workaround is to initially configure only the interface connected to the backbone network with a Class B netmask. The default route can now be added successfully since suranet-gw appears to be on the same network as the gateway. Finally, the backbone interface can be reconfigured to use subnet the netmask and the subnet interface added. It is necessary to first configure solely the backbone interface because, on Suns at least, the default route will be associated with the lowest numbered interface. Sun's official story on clients is that they must be placed on the on-board interface (unit number 0), so our gateway Suns have the connection to the backbone on the second interface. If both interfaces are configured, the default route will be set through the subnet interface and access to the suranet-gw fails. Routes to other subnets can be added by hardwiring them or added dynamically with appropriate software. We use a program called "asrd" (Automatic Subnet Routing Daemon), written by Tim Seaver at MCNC, that adds routes based on ICMP redirect messages sent to a gateway from the default gateway. This works as follows: the first time a host running asrd accesses a subnet, the packets are routed through the default gateway. The default gateway will forward appropriately and also send back an ICMP redirect packet indicating where the real gateway for that subnet is. Asrd listens for the ICMP message from the default gateway, extracts information from it, and adds a route to that subnet. Subsequent accesses to that subnet will be routed directly to the appropriate gateway. Asrd was designed with the specifics on the MCNC extended Ethernet in mind, where there are many people running different systems which understand subnet in varying degrees. A solution such as this may not be appropriate in other environments.

The gateways needed an additional piece of software, "proxyarpd", to implement the transparent network we wanted. Running this on the two interfaces made it so that arps from hosts on either network for hosts on the other would be answered by the gateway instead. Traffic sent using the information in the arp answer is silently passed through through the gateway to the other network.

Gateway hosts also provide other vital services - again with the aim in mind of minimizing intermachine dependencies. The services central to transparent networking are the Berkeley nameserver (bind) and Sun's Yellow Pages (YP). Although we, for a time, had YP and bind linked in terms of hostname resolution, bind and YP are now completely independent. Utilities that are likely to need resolution of ''outside'' host names are linked to use bind directly. The others are left linked to use YP. Implementing these name services on the gateways provides redundancy for hosts on the backbone while ensuring that all hosts on subnets receive this service. Hosts have the same network functionality regardless of their placement on the network.

# spy: A Unix[1] File System Security Monitor

Bruce Spence

*HEWLETT-PACKARD*

*Colorado Integrated Circuits Division*
Technical Computer Support Group

Colorado Integrated Circuits Division of Hewlett-Packard is a designer and manufacturer of custom integrated circuits for use within HP. The R&D Lab provides contract design services in addition to designing IC's for in-house fabrication, and Manufacturing also fabricates devices not designed in-house, so Company-wide networking is a vital way of life.

We are a very computer-intensive operation, utilizing a variety of resources. Our IEEE-802.3 coax local area network presently connects 203, HP 9000 series 300 systems; 12, HP 9000 series 800 systems; and one HP series 500 computer, all running under the HP-UX operating system. The network is also home to over one hundred Vectra PC's.

Over three hundred users, with widely-varying Unix and computer expertise, are presently using these systems for differing purposes, such as IC design, IC testing, data reduction and evaluation, text editing and formatting, graphics generation and output, reading notes, electronic mail, etc. The system is administered by five dedicated (!) Technical Specialist support personnel, with some administrative work being done by workstation owners themselves.

As with all operations that rely on computer technology in their businesses we are concerned with computer and network security. Security-related concerns fall into a number of areas:

- The sensitive proprietary nature of our IC designs and processes demands that computers and data giving access to information about designs, tools, and processes be reliably secure from non-Company personnel.
- The strongly inter-related nature of our operation and necessary open linkages to our customers, as well as within the Division, require an uncomfortable open-ness of the network and many of its systems.
- There are many opportunities for security weaknesses or holes in Unix systems.
- User awareness of security issues is usually low, as is their priority of addressing security concerns.
- Users and systems support personnel have no good, ready mechanism for easily evaluating the security state of Unix systems.

In answer to some of these concerns we developed a comprehensive Unix file system security checking program, spy, which is described below.

* * * * * *

1. Unix is a registered trademark of AT&T.

**Spy** is a non-intrusive security checking program. It performs a wide range of file system checks on HP series 300 and 800 computers running HP-UX. Nothing is modified; a report of any problems found is mailed to the mail destination specified at execution time. The program is implemented as a Bourne shell script (i.e., no Korn-shell-specific features are used). It is usually executed as root so that the entire file system can be seen, but no check is made of the invoking user. **Spy** should be owned by root.

The program is invoked by: **spy mail_dest**, and only needs to be run on the server in a diskless cluster. 'Mail_dest' may be any valid sendmail destination. Multiple destinations may be specified by enclosing a list in quotes.

**spy** requires four auxiliary files, the 'exception lists' all of which are *sed* filters:

| | |
|---|---|
| pw_filter: | allowed password-less logins |
| own_filter: | allowed system files with non-standard ownerships |
| perm_filter: | allowed system files with non-standard permission (that is, with general write permission on). |
| suid_filter: | allowed SUID-root and SGID-root files. |

The program requires these files to exist, but any or all may be of zero length. They must reside in the same directory on the master system, which may be any system reachable by *ftp* (see note on the configuration file below). These files generally need some localization.

So that on older operating system revisions we can be sure of handling NFS-mounted file systems properly (that is, **spy** ignores them), this same directory contains a copy of new *find* command binaries which understand the '-fsonly' option. **Spy** copies over for its use the *find* appropriate for the operating system on which it is running .

There is a required configuration file called 'spy.config' under root's home directory on each system running **spy**. This file contains definitions of four localizable parameters required by **spy**:

mastersys: system on which the exception files reside.

user: user via which *ftp* access to exception files is established. Ideally, for security reasons, this pseudo-user account should not yield an interactive shell and should only be used for the master file access needed by **spy**.

password: password via which *ftp* access to exception files is established; must be non-null.
masterdir: directory on mastersys in which exception files and the *find* file(s) live.

These may be in any order but must be of the form: parm=value .

* * * * * *

Briefly, **spy** performs the following checks:

- Checks the passwd file for password-less entries, non-standard UID-0 entries, ',..' entries, and blank lines.
- Checks a root access authorization file for ownership and permission. This routine recognizes a *su2*-style super-users file and a file used by a locally-developed similar program called *sudss*. (These and similar utilities give access to root privileges with either greater ease than the *su* command or with some additional measure of control.)
- Checks the entire file system for unauthorized SUID-root or SGID-other files.
- Checks system files and directories for public write permission. Since it is a special weak point, the /etc/hosts.equiv is specially checked.
- Checks system files and directories for proper ownerships.

- Checks the following device files for proper permissions: raw and block disks, swap, mem, and kmem.
- Checks ownership, permissions, and contents of root's .rhosts file, looking for non-root entries.
- Checks permissions and contents of the inetd.sec file, verifying that all services in /etc/services have corresponding inetd.sec entries, and that all services have some access restrictions (i.e., within HP at the least).
- Checks the /etc/exports file for proper ownership and permissions, and checks that no exported file systems are mountable by everyone.
- Checks for public write permission on crontabs, and checks for ownership and public write permission on all files executed out of root's crontab.
- Checks for presence of the 'securetty' file, which controls direct root login (i.e., requires the use of *su* or similar).
- Checks for any potential dial-in modems on the system; if found, they are checked for 'dialups' security.
- Checks for a globally-set umask in /etc/profile; if one is set, checks to be sure it doesn't give public write permission on newly-created files.

We are aware that certain assumptions as to what constitutes a secure system are inherent in **spy**; this seems unavoidable.

Following is a more detailed description of **spy** operations and implementation, and each of its checks:

* * * * * *

The following represent the possible output sections of **spy**. Each leading descriptive line, reproducing an output message from **spy**, is preceded by a letter indicative of the priority of problem as follows:

A: Critical, should be remedied immediately.
B: Serious, should be addressed as soon as is reasonable.
C: Minor, but should be fixed when time permits.

Following each section message is brief commentary on the security considerations of the problem, and suggestions for corrective action. Some sections also contain an implementation note in the form of an un-documented code fragment.

===============

A "Non-standard password-less logins found:"

Password-less logins that give an interactive shell (ksh, sh, or csh) allow anyone free access to the system. This bypasses the first, primary line of security, the user password. All accounts in the password file should have passwords unless there is a compelling reason otherwise. Any password-less login should have /bin/sync or some such as a shell, and should only be used for network file access. All accounts yielding interactive shells must have passwords or must have an asterisk in the password field, preventing login. Note that the password entry: ,.. is effectively the same as a null password field, as it will allow anyone to login once, and should be avoided.

To correct, star out any empty password fields or have the user enter a valid password.

A routine of the following form is used to build a list of passwordless entries:

```
cat /etc/passwd |
while pwline='line'
```

```
do
        pw="'echo $pwline | awk -F: '{print $2}' '"
        if [ "$pw" -a "$pw" != ",.." ]
        then
                continue
        else
                echo $pwline >>$temp
        fi
done
```

+++++++++++++++

B "Non-standard UID-0 logins found:"

Root capability is controlled by the User ID (UID) number, not the name 'root'. Thus, any UID-0 account has full root capability. Such accounts other than the standard 'root' should be avoided as they represent proliferation of root capability without good traceability, as the user ID is the same for all entries.

To correct, remove all UID-0 entries from /etc/passwd other than 'root', if possible. If persons other than the system manager must have root capability it is preferable to use a controlled root-access method such as the *sudss* or *su2* commands.

A routine of the following form is used to search for non-standard UID-0 entries:

```
grep "^[a-z]*:\*:0:" /etc/passwd >$tempa
grep "^[a-z]*:.............:0:" /etc/passwd >>$tempa
sed -e '/^root:/d
        /^dss:/d' $tempa >$temp
```

+++++++++++++++

A "Blank line(s) found in /etc/passwd:"

This is a most serious security hole. The first time that any user changes his or her password the blank line will be changed into a pseudo-entry that may allow anyone on the network to gain root privileges.

To correct, remove any blank lines in /etc/passwd.

+++++++++++++++

B "Non-standard dss access file entries found:"

This refers to the *sudss* controlled root access system in place on some systems administered by our Technical Systems Support Group. This warning indicates that there are some entries in the access control file in addition to those in the master file.

Corrective action is referred to the Technical Systems Support Group.

+++++++++++++++

A "Permissions are improper on root access file:"

The root access control file, whether the one referred to above or the 'super-users' file used by the *su2* command must not be publicly writable, or any user may be able to, in effect, give him- or herself root privileges. In addition, such file should not be publicly readable, or an invader can easily find out who can gain root access and, thus, where to look for user security holes.

To correct, execute a '*chmod o-rw*' or similar command on the root access control file.

+++++++++++++++

**B** "Root access file is not owned by root:"

Under the same basic reasoning as immediately above, the root access control file must be owned by root to protect it from unauthorized access.

To correct, change ownership of the root access control file to 'root' and, probably, to group 'other'.

+++++++++++++++

**B** "Non-standard SUID and/or SGID root files found:"

Generally speaking, when a binary program is executed it executes as, and with the permissions of, the executing user, independent of the ownership of the executable file. This can be changed by turning on what is called the 'set user ID' (SUID) bit of the file permissions. When this bit is set the program is executed as, and with the permissions of, the *owner* of the file. Certain system commands must run as root, so these files will be owned by root and be SUID so that they execute with root privileges. An example of this is the *passwd* command. SUID-root files do present a security hazard, however, as an intruder may be able to misuse one or be able to break out of one to gain more general root capabilities, especially if the software is not carefully written to avoid such mis-use. Thus, SUID-root files should be restricted to those required by the system or those absolutely necessary secure local programs. The files listed by **spy** in this section are non-standard SUID-root files.

To correct, un-set the SUID bit, if feasible, by executing: '*chmod u-s*' on the file. This may not be possible if the non-standard file must be SUID-root for some valid reason.

A pipeline of the following form is used to perform the search:

```
find / -hidden -fsonly hfs -type file \( \( -user root -perm -4000 \) -o
 \( -group other -perm -2000 \) \) -print | sed -f suid_filter >$temp 2>/dev/null
```

+++++++++++++++

**B** "Root directory permissions are non-standard:"

To control write access to system files the root directory ('/') must not be publicly writable.

To correct, execute: '*chmod 755 /*'.

+++++++++++++++

**C** "/etc/btmp permissions are too open:"

The /etc/btmp file logs failed login attempts. It is fertile ground for hunting for passwords, which are often mistakenly typed as logins. It should not be publicly readable.

To correct, execute '*chmod 600* /etc/btmp'.

+++++++++++++++

**C** "System files/dir.'s found with general write permission on:"

As a general rule, a file or directory should have general, public write permission turned on only if there is a need to do so. Nearly all system files should not be publicly writable, both as a matter of good practice and to prevent accidental (or intentional) unauthorized over-writing. The files/directories reported here are those that do not need general write permission turned on for any known system reason.

To correct, execute: *'chmod o-w '* on the affected files.

A pipeline of the following form is used to perform the search:

```
find /etc /bin /lib /usr/adm /usr/bin /usr/lib /usr/local/bin /usr/local/lib
  /net /usr/contrib/bin /usr/contrib/lib $roothome -hidden -fsonly hfs -type file
  -perm -002 -print | sed -f perm_filter >$temp 2>/dev/null
```

+++++++++++++++

**A** "/etc/hosts.equiv file is publicly writable:"
This can allow access with full root privileges. The hosts.equiv file lists systems which are considered 'equivalent' to the host on which the file exists. A user on a system listed in this file, who has a login on the file-containing system, can log in as him- or herself without password. This includes the 'root' user, and so represents an uncontrolled root access opportunity. If it exists this file should be writable only by root. Its use is discouraged in favor of the more selective '.rhosts' mechanism.

To correct, eliminate the /etc/hosts.equiv file or to execute *'chmod 600* /etc/hosts.equiv' if the file will exist.

+++++++++++++++

**C** "Improper user permissions:"
As a matter of good overall security, users' home directories and their .profile and .rhosts files, if any, should not be publicly writable. This is especially true for users that can gain root privileges. This is for their own protection but also makes it more difficult for an intruder to gain wide access to the system.

To correct, execute *'chmod 755'* on the user's home directory and to execute *'chmod o-w'* on the user's .profile and .rhosts as necessary.

The following code is used to check permissions on users' home directories and any .profile and .rhosts files found there, and build a list of such files:

```
awk -F: '$3 >100 {print $6}' /etc/passwd |
sed -e '/uucpp/d
     /\/tmp$/d' >$tempa
for dir in 'cat $tempa'
do
     if [ -d $dir ]
     then
          if ll -d $dir | awk '{print $1}' | grep "........w." >/dev/null 2>&1
          then
               ll -d $dir >>$temp
          fi

          for file in '.profile' '.rhosts'
          do
               if [ -f ${dir}/$file ]
               then
```

```
                    perms='ll -d ${dir}/$file | awk '{print $1}''
                    if echo $perms | grep "........w." >/dev/null 2>&1
                    then
                        ll -d ${dir}/$file >>$temp 2>/dev/null
                    fi
                fi
            done
        fi
    done
```

+++++++++++++++

C ".netrc files found:"

The .netrc file allows a user to specify system-user-login combinations for transparent access. The use of this mechanism is strongly discouraged, as it requires the .netrc file to be publicly unreadable and the user's home directory to be unreadable and unwritable for minimal security protection of the user to exist, as un-encrypted passwords exist in the file.

To correct, remove the .netrc file. If the user insists on using this file then the user or root should execute: '*chmod 600*' on it and should execute '*chmod o-rw*' on the user's home directory.

+++++++++++++++

C "Non-standard ownerships:"

With a few exceptions (which are not reported) system files should be owned by one of a selected group of users and groups. Ownership by other than this group opens the file system to increased chances of accidental (or intentional) removal or alteration of system files.

To correct, change the ownerships of reported files to standard ownerships. Generally, either 'root' or 'bin' are acceptable user owners, and 'other' and 'bin' are acceptable group owners. However, there are some notable exceptions. In particular, SUID files should have ownerships changed only with care, as some non-standard ownerships of such files are necessary (such as /usr/bin/lp).

Pipelines of the following form are used to perform the searches:

```
find /etc /bin /lib /usr/bin /usr/lib /net /usr/local/bin
 /usr/local/lib /usr/contrib/bin /usr/contrib/lib /system $roothome
 -hidden -fsonly hfs \( ! \( -user root -o -user lp -o -user bin
 -o -user uucp \) -o ! \( -group bin -o group other -o group mail
 -o -group root -o -group sys -o -group daemon \) \) -print |
sed -f own_filter >$temp

find /usr/adm -hidden -fsonly hfs
\( ! \( -user root -o -user bin -o -user adm \) -o
! \( -group bin -o -group other -o -group root -o -group sys
-o -group adm \) \) -print | sed -f own_filter >>$temp
```

+++++++++++++++

C "Root directory ownerships are non-standard:"

To control access to system files the root directory ('/') should be owned by root.

To correct, execute *'chown root /'*, and probably *'chgrp other /'* as well.

+++++++++++++++

C "Device files found with improper permissions:"
It is important that certain key disk and memory device files not be publicly readable or writable. This is to prevent uncontrolled direct access to disks and to prevent direct reading of memory, which would effectively bypass read security. The devices are, typically:

/dev/dsk/*
/dev/rdsk/*
/dev/swap
/dev/mem
/dev/kmem

Corrective action is, at a minimum, to execute *'chmod o-rw'* on the files. Mode 200 or 400 is even better.

A pipeline of the following form is used to perform the search:

```
find $devdirs -hidden \( -type b -o -type c \) \( -perm -002 -o
 -perm -004 -o -perm -020 \) -print |
sed -n -e '/\/dsk/p

        /\/rdsk/p
        /\/swap/p
        /\/kmem/p
        /\/mem/p
        /\/root/p' >$temp 2>/dev/null
```

+++++++++++++++

B "Non-root entries found in root's .rhosts file:"
The services yielding an interactive remote login shell require a valid user password for access, as do *rcp* and *ftp*. This may be selectively avoided by the use of the '.rhosts' file. System-user pairs listed in this file in a user's home directory may access ARPA-Berkeley services on that system as the user owning the file, without password. This applies to root as well. Because of potential serious security holes here, only essential root entries from local systems should be in root's .rhost file. If no file exists, no password-less access is allowed. This is the preferred configuration for root. If a .rhosts file exists for root it should contain no non-root entries, as this bypasses the principle of at least two passwords for root access, allowing a non-root user on another system to become root on the reporting system.

To correct, remove the non-root entries from root's .rhosts file.

The following fragment checks root's .rhosts file for presence of non-root entries:

```
cat ${roothome}/.rhosts |
sed -e '/^#/d
        /^[    ]*$/d' |
while hostline=`line`
do
        if echo "$hostline" | grep root >/dev/null 2>&1
        then
                continue
        else
                echo "$hostline" >>$temp
        fi
done
```

+++++++++++++++

A "Root's .rhosts file is publicly readable and/or writable:"

It is essential that all .rhosts files be not publicly writable. This is especially vital for root's, as a publicly writable root .rhosts file gives easy unrestricted root access to anyone. Root's .rhosts file, if any, should also not be publicly readable, to prevent an intruder from knowing what other systems can gain password-less root access to the reporting system.

To correct, execute '*chmod o-rw*' on root's .rhosts file.

+++++++++++++++

C "Some offered services not in inetd.sec file:"

The /usr/adm/inetd.sec file specifies the systems that are allowed access to system services via the network. If no inetd.sec file exists then no restrictions apply. All services in /etc/services should have entries in the inetd.sec file. While some 'non-standard' services, such as locally- developed ones, may not assume the proper responsibility to check the inetd.sec file, it is better (and non-harmful) to include all services. Allowed system access should be kept to a minimum. Output in this section indicates that some offered services have no corresponding entries in the inetd.sec file.

To correct, add the necessary entries to the inetd.sec file.

The following routine checks for entries in /etc/services with no corresponding inetd.sec entry:

```
netsec=/usr/adm/inetd.sec
cat /etc/services |
sed -e '/^#/d
        /^[         ]*$/d
        /qless/d' |
awk '{print $1}' |
while servline='line'
do
        if grep "$servline" $netsec >/dev/null 2>&1
        then
                continue
        else
                echo "Service $servline not found in $netsec" >>$temp
        fi
done
```

+++++++++++++++

C "Non-standard entries found in inetc.sec file:"

This indicates that some services are being offered too widely. Good practice indicates that most services should be offered only as widely as is necessary. **Spy** assumes that *ftp*, *telnet*, *smtp* (the sendmail daemon), and *install* (the ninstall server daemon) services may be offered to any HP systems. All others should be restricted to one or more specified sub-nets, such as 15.1.2.xx .

To correct, add subnet restrictions to the indicated services.

+++++++++++++++

**B** "inetd.sec file is publicly writable:"

Having the inetd.sec file publicly writable allows defeat of all network service access security by anyone so choosing. This file should be owned by root and should not be publicly writable.

To correct, execute '*chmod o-w*' /usr/adm/inetd.sec.

+++++++++++++++

**B** "inetd.sec file is not present:"

If there is no inetd.sec file, then no restrictions are placed on access to system network-based services. This file should exist as outlined above.

To correct, properly create the inetd.sec file.

+++++++++++++++

C "/etc/exports has general write permission on and/or improper ownerships:"

The NFS mount system allows a system to mount part of a remote file system on its file system. File systems can only be mounted by root, and should be mounted read-only whenever possible to protect contents of the remote file system. The system requires an entry in the /etc/exports file on the remote system (the one containing the file system to be mounted) for the file system to be mounted; if no /etc/exports file exists no remote mounts are allowed. Output in this section indicates that the /etc/exports file is not adequately protected from unauthorized removal or alteration. It should be owned by root and should not be publicly writable.

To correct, execute the following:

```
chown root /etc/exports
chgrp other /etc/exports
chmod o-w /etc/exports
```

+++++++++++++++

C "One or more file systems are being NFS exported with no restrictions:"

Entries in /etc/exports can, and should, restrict mount access to specified systems (or groups of systems as specified in /etc/netgroup). If no systems are specified then any system running NFS software can remote mount from the reporting system. Thus, all entries in /etc/exports should have some restrictions on who can mount, such as the following example:

```
/mnt1  hpfifoo
/mnt2  mygroup
```

To correct, modify the /etc/exports file as necessary to add mounting restrictions.

A pipeline of the following form is used to perform the search:

```
if [ "'cat /etc/exports | sed -e '/^#/d
        /^[          ]*$/d' | awk 'NF < 2 {print "f"}'`" ]
then
        echo "One or more file systems are being NFS exported with no restrictions."
fi
```

+++++++++++++++

C "Crontabs found with general write permission on:"
'Cron' is a system program that runs constantly in the background, scheduling jobs to be run at pre-specified times. The crontab file is used to schedule jobs for periodic execution via cron. The crontab files (under the /usr/spool/cron/crontabs directory) are created by the system with public write permission off. These files should not be directly manipulated, but should only be altered via the *crontab* command (see the *crontab*(1) man page). Crontabs, especially root's, should not be publicly writable, else anyone can add an entry to execute anything that they wish (as long as it is executable by the owner of the crontab). This is especially dangerous in the case of root's crontab, as anything executed from this crontab is executed as root, regardless of who owns the executable file. This would allow unrestricted root access by unauthorized persons. Output in this section indicates that some crontabs have public write permission turned on.

To correct, execute '*chmod o-w*' on the affected crontab files.

+++++++++++++++

B "Files found being executed out of a root crontab with general write permission turned on:"
This is really an extension of the problem with a writable root crontab mentioned immediately above. If a file being executed out of root's crontab is publicly writable then any user (or intruder) can modify or replace such file as he or she chooses, again gaining effectively unrestricted root privileges when the file is next executed by *cron*. Thus, all files being executed out of root's crontab should have general write permission turned off and should idealy have 'standard' system ownership (see above). In addition, all directories in the path to the executable file should have general public write permission turned off.

To correct, execute '*chmod o-w*' as necessary on the reported executable files and any publicly-writable directories in their paths. The following routine is used to perform the search by extracting the paths of all files being executed out of root crontabs after first eliminating certain environmental variables that may be being set on the fly:

```
cat $crontabs | uniq >>$tempa
cronexec='sed -e '/^#/d
        /^[          ]*$/d
        s/[           ]HOME=[/,a-z,A-Z,0-9]*[     ]//
        s/[           ]PATH=[/,a-z,A-Z,0-9]*[     ]//
        s/[           ]SHELL=[/,a-z,A-Z,0-9]*[    ]//
        s/[           ]IFS=[/,a-z,A-Z,0-9]*[      ]//
        s/[           ]TZ=[A-Z][A-Z,0-9]*[        ]//
        s/exec //
        s/sh //' $tempa | awk '{print $6}''
find $cronexec \( ! \( -user root -o -user bin -o -user uucp \) -o
  -perm -002 \) -exec ll {} ; >$temp 2>/dev/null
```

+++++++++++++++

C "Root has no crontab:"
This highly-unlikely circumstance does not represent a specific security problem so much as an operational one. It is usually essential for proper system operation that there be a root crontab.

To correct, create a root crontab (see the *crontab*(1) man page and the System Administrator Manual).

+++++++++++++++

**C** "Securetty file is not present:"

If the /etc/securetty file exists it prevents direct login as root from any 'port' (ttyxx, ptyxx, etc) not specified in the securetty file. The only way to login in as root is to use the *su* command after logging in as some other user. Best practice would have this file exist as a null file, thus preventing *any* direct login as root. This again is in accordance with the principle of requiring at least two passwords to gain root access.

To correct, execute the following:

```
touch /etc/securetty
chmod 600 /etc/securetty
```

+++++++++++++++

**A** "Possible modem, unsecured, found:"

Output in this section indicates the possible presence of a dial-back modem, without adequate security controls, on the system . All dial-back modems should have secondary security. One form of such security is some form of authentication of inbound users. Another possible form of such secondary security is the use of the 'dialups/d_passwd' mechanism provided as a part of standard HP-UX (see the dialups(4) man page for details). An additional security measure after some number of unsuccessful access attempts in a given time period would be to shut off modem access or, preferably, to add timeouts of increasing length to modem access.

To correct, implement secondary access security controls on the affected modem(s).

+++++++++++++++

**B** "Excessively open umask setting in /etc/profile:"

This indicates a problem that is likely the root cause of some permission problems reported earlier by **spy**. If the system umask is set to 000 or 001 or 004, then newly-created files will be publicly writable for any users who do not set their own umask. This is generally undesirable, and is especially so in the case of root.

To correct, execute: '*umask 002*' or some more restrictive value in /etc/profile. Note that it is also good policy to have root's .profile execute a '*umask 022*' or similar.

The following is used to check for an appropriate global umask setting:

```
file="/etc/profile"
if umsk='grep 'umask[          ]' $file' 2>/dev/null
then
        mask='echo $umsk | awk '{print $2}''
        perm='expr substr $mask 3 1'
        if [ "$perm" -lt 2 -o "$perm" = 4 ]
        then
                echo $umsk >$temp
                echo "Excessively open umask setting in $file:"
        fi
fi
```

* * * * * *

There is a related shell script of interest called: **tighten_sec**. It is an analogue to **spy** which actually corrects *some* of the security problems encountered. Only file ownerships and/or permissions are changed, and those as minimally as possible. No file contents are changed. The script uses the 'own_filter' used by spy as well as the same three localizable variables (see above). If invoked with the '-v' option it will mail a report of actions to the destination specified following the -v. Our experience with this script is that it will correct up to half of the exceptions typically encountered in a first-time run of **spy**.

* * * * * *

Our experience indicates that when first run on a mature, non-centrally administered HP-UX engineering system **spy** will report somewhere on the order of a hundred exceptions, with five or more being Priority A (critical). In preparing for an internal audit early this year (1989), by a combination of using the **tighten_sec** script and manual work we were able to easily reduce the incidence of critical exceptions by a factor of fifty, and reduce the overall rate of exceptions by nearly an order of magnitude.

While our security concerns haven't magically evaporated, they have been considerably reduced by the ongoing use of **spy** .

* * * * * *

# Op: A Flexible Tool for Restricted Superuser Access

*Tom Christiansen*

CONVEX Computer Corporation
POB 833851
3000 Waterview Parkway
Richardson, TX 75083-3851

*{uunet,uiucdcs,sun}!convex!tchrist*
*tchrist@convex.com*

*ABSTRACT*

The *op* tool provides a flexible means for system administrators to grant trusted users access to certain **root** operations without having to give them full superuser privileges. Different sets of users may access different operations, and the security-related aspects of environment of each operation can be carefully controlled.

One sure way to render a UNIX system unstable is to distribute the **root** password to everyone who thinks they need it. Well-meaning and experienced though these people may be, they will inevitably introduce anomalies into your system that will cause it to malfunction in mysterious ways. You may spend hours or even days trying to determine what was changed, by whom, and for what reason. This problem occurs even when all parties involved are experienced system administrators.

Furthermore, large sites often have computer operators who attend to the routine tasks of system administration, such as dumps and restores, tape handling, system shutdown, and so on. These people may not be sophisticated and you may not wish them to have complete system privileges to do their jobs. Denying the superuser password to your coworkers or management is difficult, if not impossible. This difficulty is particularly true in a technical environment where programmers may be competent, but insensitive to the management of a complex system.

The *op* program, a standard utility provided by CONVEX, is specifically designed to address this problem. The *op* program gives the system manager a means to grant a user or group of users limited access to specific superuser commands without granting access to all superuser privileges. Careful control of the environment provides both flexibility and security.

To set up your system, the *op* program, you begin by finding out the specific tasks for which system privileges are deemed necessary when a user requests the superuser password. Complete access to every command on the system is usually not required. Using *op*, the system manager can designate a set of privileged commands and access lists for these commands; the system manager, in effect, can grant limited system privileges beyond those normally available to a normal user without giving away full superuser privileges.

The *op* program is not interactive; it functions as a prefix command, similar to *time* or *nice*, whose side-effect is to alter the user's environment in some fashion. The functions (or mnemonics) understood by the *op* program are listed in the configurable ascii data file */etc/op.access*. This file describes what commands can be performed by the *op* program, how they are to be performed, and who is allowed to perform them. For security reasons, this file should be owned and readable only by the superuser. Each invocation of *op* is logged using *syslog*(3) with the LOG_AUTH facility class.

The restrictions can be made as tight as each site demands, as determined by the system administrator who customizes the *op.access* file. This file contains a mapping of mnemonics, or operator functions, to the full pathnames of programs that should be invoked and the arguments that are allowed, if any. The arguments to the executed program can be a combination of literal and variable arguments, and restrictions can be placed on which values are valid substitutions for the variable arguments. Because some syntactic checking of the command arguments is possible, running commands under *op* can be safer than running them directly from a superuser's shell. This safety feature can prevent pitfalls like accidentally transposing the file system and tape device arguments to the *dump* program (eg. `dump 0uf / /dev/rmt16`), which would destroy the file system.

The following set of attributes can be controlled for each mnemonic by the *op* program:

- the user id to set
- the group vector to set
- the directory to *chdir*(2) to
- the root directory to set with *chroot*(2)
- the umask to set
- a list of groups allowed to execute this function
- a list of users allowed to execute this function the superuser)
- the range of valid arguments for the command, both in number and value
- any environment variable settings

The fields of the entries in *op.access* are separated by white space. Each entry may span several lines and continues until the next alphanumeric string is found at the beginning of a line (which is taken to be the next *mnemonic*, and thus the beginning of a new entry). Comments may be embedded beginning with a # character. Each entry in *op.access* has the following form:

*mnemonic* *command* [ *arg* ... ] ; [ *option* ... ]

where the fields are interpreted in the following manner:

| | |
|---|---|
| *mnemonic* | a unique, alphanumeric identifier for each operator function. |
| *command* | the full pathname of the executable to be run by *op* when the associated *mnemonic* is chosen. |
| *arg*(s) | any arguments, either literal or variable, needed by *command*. Literal |

arguments are simply specified directly, like specific command options (**0Gun**) or files (**/dev/rmt20**). Variable arguments are specified here as **$1, $2** ... **$***n*; these are described more fully in the options section below. **$*** indicates any number trailing arguments.

*option*(s) a set of optional parameters to specify settings or restrictions for the particular *mnemonic*, define variable arguments specified for the *command*, and define environment variable settings. Options are separated by white space and are of the form *keyword=value*. The absence of a specific option means the default is sufficient. The *value* can be a single value or a list of values separated by commas, where appropriate. There should be no white space in each element of the *value* string unless quoted. The *keyword* is any of the following types:

**uid** Set the user id to the value specified. The value can be a numeric user ID or a login name. The default is **root**.

**gid** Set the group ids to the values specified. Each value can be a numeric group ID or a group name.

**dir** Change the current working directory to the path specified.

**chroot** Change the root directory to the path specified using *chroot*.

**umask** Set the file creation umask to the octal value specified. The default is to set it to **022**.

**groups** Allow any user who belongs to a group listed here to execute this *op* function. The default is not to allow any specific group.

**users** Allow any user listed here to execute this *op* function. The default is not to allow any specific users. You may use the regular expression **.*** to indicate that all users may use this mnemonic.

$*n* defines the *n*th variable argument specified in the command *arg* list. The value for this type may be a comma-separated list of regular expressions using *regex*(3). option defines the range of values allowed for the variable arguments. A variable argument specified as a command *arg* but not described in the *options* section may take on any value. If an argument does not match any of its permitted values, then a diagnostic is printed and the command is not executed.

$* is used in the *options* section to place restrictions on the trailing arguments specified as $* in the *args* section. If any of these (possibly many) arguments do not match, then a diagnostic is printed, and the command is not executed.

$*VAR* where *VAR* is the name of an environment variable. The specified environment variable is set to the value given before the command is executed. As a special case, simply using $*VAR* with no = part (as in **$USER**) means that this environment variable is inherited unchanged from the caller's shell.

There can also be a special entry in the file beginning at the first non-comment line that can define default values to override the builtin defaults listed here, yet still be overridden by any entry that wants to redefine any of the keyword fields described above. It should have the following format:

**DEFAULT** *keyword_option ...*

where *keyword_option* is a *keyword=value* strings mentioned above under *options*.

It should be noted that if any regular *mnemonic* entry defines its own *option*, the value given for that entry must explicitly include the item from the DEFAULT line if the default value is to be included. That is, the *options* definitions completely override any defaults; they do not add to them. In this way, if a value specified on the DEFAULT line for **users** or **groups** (for example) needs to be ''erased'' without redefining new values (that is, we want no users or groups to be allowed to run this mnemonic), then the default value must be overridden with nothing (as in **users=**). For the **users** and **groups** fields, such a null setting has the effect of setting the list of allowable users or groups to be empty. For the other keywords (**uid, gid, dir, chroot**, and **umask**), a null setting leaves that attribute as it is upon invocation of the *op* program, overriding any defaults.

This file format may seem complex at first glance, but is actually intuitive and flexible. An example *op.access* file might look like:

```
# first, define the site defaults we want to use here
# we would like the people in 'operator' group to be able to execute
# almost everything, so it is easier to put it here than on every line...
# set up default envariables
#
DEFAULT groups=operator $USER $TERM $PATH=/usr/ucb:/usr/bin:/bin
#
#   find out who's filled up the disk; anyone may do this
#
full        /usr/etc/quot $1; users=.*
#
#   filesystem backups
#
daily       /etc/dump 5Gun $1; $1=/,/usr[0-9]*,/project
weekly      /etc/dump 0Gun $1; $1=/,/usr[0-9]*,/project
#
#   tape handling commands
#   must include 'operator' if we want them to be allowed as well
#
tape        /etc/tpc $1 $2; groups=tapeopers,operator users=boss
          $1=enable,disable,stop,restart $2=all,unit[01]
#
mounted     /etc/tpc mounted unit$1 $2; $1=[0-3]
#
#   taking the system down
#   $1 shows a good use of regular expressions;
#   $2 can be anything, but is required; no instant shutdowns
#
shutdown    /etc/shutdown -h $1 $2; $1=+[1-9][0-9]*,[0-9]*:[0-9]*
reboot      /etc/shutdown -r $1 $2; $1=+[1-9][0-9]*,[0-9]*:[0-9]*
#
#   start up disco daemon
disco       /etc/opbin/start_disco; uid=disco gid=proj dir=/scratch
              umask=027 groups=geo,disco users=snoopy,linus
              $USER=disco $SHELL=/bin/shell
#
#   let certain people mount and unmount the removable drive
#
rdsmount    /etc/mount $1 $2; groups=operator,swdev,disco users=bob,steve $1=/dev/dd0[a-g] $2=/.*
rdsumount   /etc/umount $1; groups=operator,swdev,disco users=bob,steve $1=/dev/dd0[a-g]
#
#   allow operators to give files away; notice that they
#   they must give at least two args, but may give more
#
chown       /etc/chown $1 $2 $*; $1=[a-z0-9][a-z0-9]*
#
#   permit development personnel to run install
#
inst            /usr/bin/install -o root -g system $1 $2; groups=devel
                  $2=/bin,/usr/bin,/usr/ucb,/usr/new,/usr/local
#
nfsmount        /etc/mount -o timeo=100,hard,intr $1 $2; groups=devel,operator
                  $1=\([a-zA-Z0-9_]*\):\(.*\) $2=/remote/\1\2
```

Some example command lines using *op*, given the above *op.access* file, might be:

```
% op full /usr1
% op weekly /usr1
% op tape disable unit0
% op reboot 17:30 "We have to fix our network."
% op disco
% op rdsmount  /dev/dd0c  ~/mystuff
% op mounted 3 8688
% op chown jim  /tmp/bill/*
% op inst less  /usr/local
% op nfsmount  convexs:/usr/src  /remote/convexs/usr/src
```

Note that the following commands would not work because they would not match the back-reference specifications in the *nfsmount* mnemonic:

```
op nfsmount convexs:/usr/src /remote/foobar/usr/src
op nfsmount convexs:/usr/src /remote/convexs/src
```

In summary, the *op* program allows the system manager to give out limited system privileges without compromising the **root** password. The system can be easily tuned to the needs of a specific site. The environment in which these commands execute can be tightly controlled and their arguments checked for valid values. For security reasons, a log is kept of all commands run. Careful application of the *op* program can result in a stabler system.

# Filesystem Backups in a Heterogeneous Environment

*Ken Montgomery*
*Dan Reynolds*

The University of Texas System
Center for High Performance Computing
Balcones Research Center
10100 North Burnet Road
Austin, Texas 78758-4497

June 29, 1989

*ABSTRACT*

Performing filesystem backups is a common activity for Unix system administrators. In a heterogeneous computing environment where a large number of distinct Unix systems from different vendors are used, the problems multiply: some systems have inadequate tape hardware for convenient, reliable backups. Moreover, the software to perform filesystem backups differ, even among Unix systems. To solve these problems, a modified *ftp*(1) client is used to move the output from *dump*(8) across the local network to a fileserver machine.

## Introduction

The Center for High Performance Computing was established in 1985 by the University of Texas System Board of Regents to provide high performance computing services for the thirteen component institutions of UT system. The original computer systems consisted of a Cray X-MP/24 supercomputer running the COS operating system, an IBM 4381 running MVS with DF/HSM for file service and a VAX 8600 running VMS as an access server and telecommunications front-end. In late 1987, with the growing popularity of the Unix operating system for supercomputer systems, the UT Board of Regents approved acquisition of a second supercomputer to run this operating system. Cray Research was awarded a purchase order for an EA X-MP/14se and this system was installed in November 1988 running the UNICOS 4.0 operating system.

At the same time, the VAX 8600 access server had become terribly saturated and it was clear that a more powerful system was needed. It was deemed desirable to run the Unix operating system on this machine since the supercomputers it would serve in the future would be Unix platforms. A Convex C-120 mini-supercomputer was purchased and installed in September 1988. Other Unix systems for high-performance graphics, text processing, and staff use appeared in the form of several Sun workstations with associated fileservers, an Ardent Titan, and a Silicon Graphics workstation.

Consistent administration of so many different Unix systems is difficult since the vendor-supplied utilities differ. Finding common solutions to user validation/devalidation, filesystem organization, maintenance of locally-supplied software, and filesystem backups remains a recurring problem. In the area of filesystem backups, we were able to devise a workable solution common to all the systems.

## Hardware

The smaller systems (Suns, Ardent, and Silicon Graphics) typically use 60 MB tape cartridge drives for backups. For backing up a few megabytes of data, this equipment is adequate but as the size of the information store attached to these smaller machines increases (the Sun fileserver has 800 MB currently and is scheduled to be upgraded to 2.5 GB shortly), the data transfer rate and storage capacity of the native backup hardware is hopelessly inadequate.

Like its smaller brethren, the Convex C-120 is sparsely supplied with tape backup hardware: the single 1600/6250 cpi nine-track tape drive is rated at only 50 ips. Taking a full backup of the 3.8 gigabytes of disk storage on this machine is time-consuming. Furthermore, failure of this one drive constitutes a single point of failure which could be unfortunate. Only the Cray X-MP/14se has access to anything approaching adequate backup hardware: four IBM 3420 drives and six IBM 3480 cartridge drives. However, it must share this access with the X-MP/24 and the IBM 4381.

**Software**

The classical Unix mechanism for filesystem backups rely on *dump*(8) and *restore*(8). *dump* opens the block special device containing the filesystem, scans the directory tree, and then writes the directories and regular files to the backup medium, usually on-line tapes. *restore*(8) reverses this procedure, reading the dumped information, and reconstructing the filesystem as it appeared at the time of the dump. It is also possible to incrementally dump only those files and directories which have changed since the last full dump of the filesystem.

Since many of our Unix machines lack adequate backup hardware, it was necessary to find a substitute for backups to directly-attached tapes. Furthermore, we wanted to find a common solution to this problem on all of the Unix machines. We examined the possibility of using *rdump*(8) to perform backups across the local network but decided that this mechanism was too slow for large backups. Furthermore, we already had an operative fileserver system in place in the form of an IBM 4381 with 25 gigabytes of online disk storage space, excellent tape handling facilities in the form of 3480 cartridge drives, and automatic disk-to-tape migration software (DF/HSM). Unfortunately, this system is not Unix-based; however it does provide FTP access through ACCES/MVS. If we could tie the output from *dump* to *ftp*, the backup problem would be solved.

Using the vendor-distributed *ftp* clients turned out to be a problem. Capturing the output from *dump* was no problem but getting that output into *ftp* for transmission to the fileserver was tricky. The dump files can be quite large and there is no simple way to reliably recover from errors using the standard client. Our solution was to implement a specialized FTP client which accepts the file to be transmitted from standard input (and places the received file on standard output) with larger internal I/O buffers to keep network transmission rates as high as possible. Now, it is a simple matter to pipe the output from *dump* into *pftp* and store the local filesystem image on the fileserver. Once deposited on the fileserver, the mass storage management software (DF/HSM) eventually migrates the files to 3480 cartridges for long-term storage.

A typical schedule for taking backups is followed: once weekly, a full filesystem dump is performed while incremental dumps are taken on the other days of the week. To facilitate this scheduling, a backup sequencer which interfaces to *dump* and *pftp* was written. When invoked by the operator, *backup*(8) reads a list of filesystems to dump, decides whether to perform full or incremental dumps, and then executes *dump*. *backup* also handles interfacing to the tape subsystem on each machine: the smaller machines lack sophisticated tape handling routines while the Cray and, to a lesser extent, the Convex support automatic volume recognition, assignment by volume serial number for labeled tapes, and so forth. The capabilities of the systems differ so widely that the only practical solution was a machine-dependent section in *backup* to handle each particular Unix machine.

Normally, our site performs full dumps to online tapes while incrementals are transferred across the local network. In the case of full backups, the systems are shutdown into single-user mode and the target filesystems are dismounted, with no user access during backups. This conservative policy was deliberately chosen: while it is possible to back up a mounted filesystem, it is impossible to guarantee that the backup so taken will be valid. Users can manipulate the contents of the filesystem such that *dump* can write backup tapes that *restore* cannot reload successfully. Making the various systems unavailable to users for the purpose of taking complete backups is not popular, but having reliable backups for catastrophe recovery is an absolute requirement.

*backup* is configurable so that the dump type and frequency can be varied according to the site's requirements. Note that it is possible to perform full dumps to the local fileserver using this software. Assuming the client system has an intact root filesystem and that *restore* and pftp are available, it is possible to reload complete filesystems from the fileserver. We chose to continue using on-line tapes since reloading the filesystems locally from the backup tapes is faster, particularly on the Cray and the Convex.

**Outstanding Problems**

The backup sequencer and its support utilities have proven to be useful: the software is popular with the local operations staff, as the procedure is totally automated for incremental backups with no human intervention necessary except for starting the backup process (even this could be automated through *cron*(1) but our operators like starting it up so they know backups have been run!). Even full dumps are less painful for the operators since the backup sequencer specifies the filesystems and the required tape VSNs, only requiring them to mount the tapes in the correct order.

Some problems, nonetheless, remain. First, as a supercomputer site, our Unix systems are required to deal with very large user files, on the order of one to several gigabytes. Currently, we do not back up such files at all: the number of tape reels/catridges being too large and the time required to back up even one 1-gigabyte file being 30 minutes to an hour. Limitations in the fileserver software plus low network transmission rates (200 to 300 Kbytes/second even over HYPERchannel) preclude sending extremely large files to the server. In the future, faster peripheral devices with larger capacities are needed to deal with the larger data files that supercomputer users routinely manipulate.

Second, restoration of backed-up files remains a manual enterprise. We currently have no automated mechanism for restoring files which have been inadvertantly deleted by users. Administrator intervention is required and the process is long and tedious as multiple incremental backup sets have to be searched for the correct version of the requested file.

## Availability

Persons interested in acquiring this software should contact one of the authors by electronic mail:

Ken Montgomery <kjm@hermes.chpc.utexas.edu>
Dan Reynolds <dan@hermes.chpc.utexas.edu>

or by U. S. mail at the address listed on the first page of this document.

## Conclusion

Traditionally, supercomputer centers have found it necessary to operate fileserver machines as support systems for the supercomputer(s). Supercomputer disks have large capacities and high transfer rates but are terribly expensive. Dedicated fileservers can dramatically expand the available information storage capacity much more cheaply and also provide archival storage to off-line media. But to date, none of these systems are Unix-based and, with the advent of supercomputers running the Unix operating system, inter-operability between fileserver and supercomputer client becomes more difficult.

What is needed is a File Administration System (FAS) built on top of the Unix operating system which can deal with the tremendous demands for information storage capacity which modern supercomputers generate. One of the functions that a File Administration System should provide is file backup. Instead of requiring the client systems to perform their own backups, this activity should be off-loaded to the FAS where state of the art high-performance backup devices (tape drives, VHS cassettes, optical drives, and so forth) can be provided. Using this prosaic client/server model allows for economy of scale: the supercomputer system is freed from the mundane task of filesystem backups to concentrate on numerically-intensive computing while the file administration machine can be optimized to drive these high-capacity devices at near rated speed.

---

Unix is a registered trademark of AT&T Bell Laboratories. Cray, COS, and UNICOS are registered trademarks of Cray Research, Inc. IBM and MVS are registered trademarks of International Business Machines. VAX and VMS are registered trademarks of Digital Equipment Corporation. Convex is a registered trademark of Convex Computer Corporation. Sun and Sun Workstation are registered trademarks of Sun Microsystems, Inc. Ardent and Titan are registered trademarks of Ardent Computer Corporation. HYPERchannel is a registered trademark of Network Systems Corporation.

# A better dump for BSD UNIX

Paul W. Placeway
BBN Systems and Technologies Corporation

July 28, 1989

**Abstract**

Backing up disk with vanilla dump can be a slow process; backing up 13 Gig of disk can be a *really* slow process. We changed 4.3 dump to incorporate a number of improvements, which make it faster to a local tape drive, faster over the network to remote tape drives, and more tolerant of errors. Our dump runs on number of different BSD-ish machines. Speeding up the code is discussed in detail, chronologically.

## Introduction

Our installation at the Ohio State has a large number of machines: 20 Sun servers, 5 HP servers, 5 Pyramids, an Encore, and a Butterfly, totalling over 23 Gigabytes of disk. Backing all of this up is no small task[8]; when we had only 13 Gig, our operators were spending about 40 hours each week, just spinning tapes. We were getting more disks, and knew this would only get worse with time. We observed that when backing up all of these nice fast machines, the tapes were spending most of the time just sitting still, waiting for dump to feed them more data. Clearly we needed something a bit faster than the dump our vendors provided.

We had a number of goals in mind for this project: dump must be faster when writing to a local tape drive, much faster when writing to a remote tape drive across the local area network, much more tolerant of errors, and be portable to as many machines as possible. Further, we had to retain compatibility with all of the vendor versions that we were replacing, so that any system could be restored using the `restore` program provided on the vendor boot tapes. Because of these requirements, we choose to speed up dump, rather than replacing it with something else.

## Speeding up Dump

In order to tune any piece of software, one must first understand how it works, then identify the pieces of code that make it particularly slow, and finally tune only them [1]. Sometimes the algorithm needs to be replaced, other times the general methods are fine, but the implementation can be improved upon. It is difficult to express how necessary a careful analysis is prior to any "hacking" in order for the tuning process to be successful, rather than wasted effort.

### How Dump Works

The BSD 4.2 Dump program is a very straight-forward program. Dump searches the disk, reads blocks from the disk until it has a full tape record, writes that record to the tape drive, then goes back to reading. Interspersed with the content data blocks on the tape are so called special records, which contain extra information about the dump and the files for restore. There are several types of special records, including beginning and end of tape, file allocation maps, and other file-system specific things[4]. 4.2 dump is simple, and unfortunately also quite slow.

In order to speed up dump, Don Speck of Cal-Tech had a really good idea: overlap reading from the disk and writing to the tape. He accomplished this by having dump fork off a number of slave processes (usually three) that read commands from the master process through a pipe. The Master traverses the file system, and gives each slave a map of locations to read from the disk, followed by a variable number of special records (tape header, I-node, I-node map, etc.). For each special record, the location map has the location marked as zero, and the length set to one; the special record is inserted into the tape record verbatim, at the tape record offset corresponding to the location map mark.

Each slave collects commands from the master, reads the disk blocks into it's tape record buffer, requests a lock for the tape drive, writes it's tape record to the tape, and then goes back to reading new commands from the master.

To avoid scrambling the data on the tape, use of the tape drive must be controlled by some type of acquire and release mechanism. In 4.3 dump, the master creates a pair of files in /tmp, opens them, and then unlinks them, prior to spawning off the slave processes. The slaves use the file descriptors to implement semaphores, using *flock* to implement the the acquire and release.

### Our changes for local dumps

Speck's general algorithm for doing high speed dumps is very good, causing a TU77 to stream on a VAX 11/780[7], and running about twice as fast as the 4.2 dump on a Sun 3/180, dumping at between 220 and 250 K/sec. Since our hardware is much faster than 780s, we felt that the systems were capable of still better performance.

We made a number of changes to the dump code to achieve better performance:

- The locking scheme was replaced with one using pipes.
- Each slave writes more than one tape block during its turn.
- Special records are read from the master as soon as possible, to keep the master from blocking.
- Access to the disk is controlled, to reduce disk head motion
- Reads are sorted to further reduce disk head motion.
- Reads of several close disk blocks are combined to into one large read (on some machines).
- Unnecessary subroutine calls are eliminated form the disk reading loop.

**Locking.** We wanted to use this dump on all of our machines. Unfortunately, the *flock* calls did not work correctly on our Pyramid systems. In order to simplify things, this was replaced with a ring of pipes between the slaves, writing a character token on the pipe to the next slave as the *V* operation, and reading the token as the *P* operation.[6] In Sun and Pyramid's versions of UNIX, this proved to be slightly faster, although Mr. Speck tells me that using pipes are, in fact, much slower on 4.2 Vaxen.[7]

**More tape blocks per slave.** After noticing how long the gap between writes to the tape could be, and doing some testing of the disk and tape drives, the slaves were changed so that each reads more than one tape record worth of disk, then acquires the tape drive and writes these records. The actual number of tape records each slave handles is usually between 4 and 8. This almost doubled the throughput of dump. Having each slave handle more than about 8 tape records, on the other hand, makes dump run slower, since the total system is larger than the system physical memory and parts of it must be paged in and out.

**Revised handling of special records.** I then noticed that the slave read and followed the commands from the master in a single loop. This was less than optimal, because if the last disk block on the tape record was a special record, the master would block on the pipe until the slave got around to reading the special record. This slowed down the next slave, since the master would not be ready to give its new commands until after the previous slave started writing to the tape. This results in the second slave not being ready to write to the tape drive soon enough to keep the drive streaming, and a 600 ms. tape rewind delay.

In our dump, the slave reads all the special records before doing any disk processing. This allows the master to get on to the task of collecting commands, avoiding the above scenario.

**Disk Speed and Head Motion.** I then extensively instrumented dump to reveal what operations were taking the most time. This showed that the disk drive was the bottleneck. We were somewhat surprised at this, as common knowledge says that disks are faster than tapes. Additional speedups seemed at the time to be quite difficult to obtain, as the disk reading code was fairly efficient already.

The reading code was instrumented even more, to discover if there were any patterns in the disk requests that could be exploited.

At this point, several slaves were free to make disk requests at once, possibly causing excessive disk head motion. To reduce this, a second token lock was added to allow only one slave at a time access to the disk drive. This increased the throughput by only a small amount, but was necessary to fully capitalize on the next step.

**Sorting Reads.** To further reduce disk head motion, the protocol between the master and slave was changed so that tape record position was explicit in each command. The commands could now be sorted, reducing disk head motion to a minimum for each slave's assignment. This proved to be a big win, giving a 20 to 40 percent improvement in speed.

**Combining Reads.** In the process of working on the sorting code, I added printf statements to tell me which blocks were being read and how long each was. I noticed that very often dump would read a lot of fragments that were close together on the disk, often separated by only one fragment of free space. It seemed to me that more speed could be gained by combining these reads into one larger read.

I started testing disk controllers to find out more about the raw performance of each. This proved to be fascinating, as different controllers responded in widely different ways. The most extreme example is a Sun 3/180 server, running Xylogics 450 controllers to Fujitsu Eagle disk drives. In these, raw reads of between one and 24 blocks all take the same amount of time (constant time, *not* proportional to the read size). Most controllers have a minimum size, below which reads have too much overhead to be efficient. For each controller there is an optimal read size which gives the best combination of throughput and memory used. For almost all the controllers I tested, doing reads of 1 or 2 blocks each gives less than half the throughput of using somewhat larger reads.

The generalization that dump uses is that there is some size at which reads are most efficient, as long as no more than a third of the combined read is wasted space. This optimal size is, of course, dependent on the system, disk controller, and disk types. The correct value must be chosen by hand for each different configuration.

Before a read is done, dump tries to fit the current read request and as many successive requests as will fit into a buffer of optimal size. If more than one request will fit, a single read which spans the group is done into a separate buffer, and the useful data is then copied to the tape record buffers. If only the first block will fit, then it is simply read as before. This was good for a 20 to 30% improvement in throughput.

As dump was in final testing just prior to installation, a fatal flaw was discovered on our newest disk subsystems on the Sun-3 servers (one Xylogics 753 controller running two Fujitsu 2372 disk drives per Sun). with 2-1 interleaving). Every time a combined read was performed, only the data in the first block was valid, all others were garbage. The result was a garbled and useless dump tape. Optional testing code to assure the reliability of combined reads was added, and all of the systems were tested using this. This code is enabled only during the testing process of porting dump to a new machine. A trial dump should always be done with this code enabled before installing dump with the code to do combined reads.

**Reducing Subroutine Calls.** In vanilla dump, disk blocks are read using a function (`bread()`), which tries several times to read the block and correctly handles partially successful reads. This is reliable, but has high overhead. In the new code, an attempt to seek and read from the disk is made in-line first. If this does not completely succeed, `bread()` is called to reliably read the blocks. This proved to be more of a performance win than expected, giving between 50 and 75 K/sec increase in speed (10 to 20%).

**Local Speed Results.** The final results are quite good. On our Pyramid 98x dual processor, a level 0 dump runs at between 450 and 500 kilobyte per second to a Kennedy tape drive running 110 in./sec. at 6250 BPI. This drive maxes out at 500K/s in full streaming mode. For comparison, test dumps to /dev/null give 550K/sec. The Suns could potentially dump even faster (750K/sec to /dev/null), as the disks and controllers are faster, but the tape drives are slower, maxing out at 420K/sec. Dump now runs them in streaming mode almost constantly, generally taking only two to four streaming-botch rewinds per tape.

## Remote Mag Tape Dumps

As most of our backups are done across the Ethernet to tape drives on remote machines, we also wanted to speed up remote dumps as much as possible. Using the vendor-supplied version, a dump of a 250 Meg partition, over 90% full on a Sun 3/180 to a locally attached 1/2 inch, 6250 BPI tape drive took less than half an hour; not great, but not bad. On the other hand, a remote dump of a similar partition, across only a local Ethernet, took about 1 1/2 hours. Worse yet, a remote dump of a similar partition spanning two Ethernets connected by a Proteon ring network with slow packet routers took up to 8 hours to complete. Clearly rdumps this slow could not be tolerated in a large installation with over three times as many disk servers as tape drives.

The vanilla remote tape protocol is a very straight-forward implementation; the local dump sends one command (open, read, etc.) at a time to the remote *rmt*, which processes each request and sends back its result. So, in order to write to the remote tape, each tape block that dump produces is transmitted through a TCP stream to the (remote) rmt program, which reads the block from the network, writes it to the tape drive, collects the result of the operation, and sends it in a packet back down the stream. The problem is that the local dump waits for each response before sending the next, so each block written takes the time to encode and transmit the request, plus the network latency time, the time to process the request, the time to encode and transmit the response, and the network latency again. We were getting killed by network latency on our cross-network remote dumps.

Since any error at all causes dump to give up on the current tape and re-write it, the time spent sending acknowledgments back from the remote tape was a waste of resources: only an error in writing would require a response from rmt. Thus, the obvious solution was to extend the rmt protocol to add a streaming-mode write.

## Our rdump and rmt

Our protocol extension adds a new state[1] to the rmt protocol. After opening the rmt, telling the rmt to open the tape drive, and discovering that it has this protocol extension, dump tells rmt to begin streaming write mode, where each write is *n* bytes long. Then dump sends each tape block to be written, prepended with a small message saying that this is a block. When dump is ready to close the tape, it sends an end-of-stream message, telling rmt to go out of streaming mode, and an ACK is sent back by rmt if no errors occurred while doing streaming writes, or a NAK if an error occurred. If an error does occur while writing, rmt immediately sends a NAK back to dump, and ignores all further tape blocks until and end-of-stream message is sent. Meanwhile, dump receives the NAK and responds with an end-of-stream message, reads the second NAK, and goes into its tape error dialog with the operator.

Because other programs also depend on rmt (notably including **restore**), the new rmt had to announce its presence in a way transparent to programs expecting the old one. This proved easier than expected. Rmt sends result codes encoded as printable ASCII strings. On the client side, these are converted back to numbers using **scanf**. Most directives return the result of the corresponding

[1] Classic rmt protocol defines two states: either the device is closed, or it is open

system call. Since file descriptors make no sense over the network, however, the *open* directive returns zero when successful. Our new rmt sends back the code "00", which scanf happily converts to zero, just as always. Dump looks at the string returned first, however, and detects the difference.

Even if dump detects that it is talking to an original protocol rmt, it still gets an improvement in speed by delaying reading the ACKs until they actually appear, thus gaining the advantages of a windowing protocol. Write commands are sent back-to-back, so the delay is eliminated.

Using the new `rmt` with the new **dump** gives a 200% to 400% speedup over the original programs across one Ethernet, and dumps through the Proteon routers are now no slower than the Ethernet-only ones. At this time the bottleneck is the speed of TCP. This, fortunately, can be improved on.[2]

## Fault Tolerance

A good backup system must be as robust as possible. There are several ways in which this dump is more robust.

Dump is notoriously bad at figuring out how many tapes a backup would take. The old dump calculated how much data would fit on a tape by estimating the size of a tape block plus inter-record gap (in tenths of an inch), and dividing the length of the tape by this to figure out how many blocks would fit. This proved to be a problem for us, as telling dump that a 2400 foot tape was really 2200 feet would still run past the end, but telling dump 2100 feet left quite a bit of blank tape on the reel.

Our dump does essentially the same calculation, using floating-point to assure accuracy of the calculations. We also re-calibrated dump's idea of the length of a block by measuring a drive with a known constant IRG. Dump now comes within 20 feet of the end of tape.

Previously, when an error occurred on a local dump, the only option was to mount a new tape and start again from that tape. The situation was even worse if the network died while doing a remote dump – rdump would just exit, and the entire dump would have to be repeated. This often resulted in scenes of operators huddled around the console of one crash-prone machine, dumping to another crash-prone machine, counting down in unison the last five minutes of a four-hour dump which had already been repeated twice unsuccessfully.

The solution to this has three parts. First, we combined dump and rdump, so that a single flag dictates whether dump sends output to a local tape drive or to `rmt` on another machine. Second, on a tape error, dump can now switch to a different tape drive and machine (and even between local and remote dumping thanks to the first change) when starting the tape over again. Third, network failure is considered a type of tape failure, and dump will try to re-open the connection (possibly to a different machine) and restart that tape, rather than aborting the entire dump.

## Portability

We have many different kinds of machines that use the BSD fast file system[3]: Sun 3s, 4s, and HP 9000 series 300 workstations, Pyramids, an Encore Multimax and a BBN Butterfly. Naturally, we wanted to run this new dump on as many different machines as possible. Also naturally, every manufacturer has changed their systems to be slightly different than Berkeley.

Much time was spent comparing the source code of the different manufacturer's versions of dump, finding the differences, and figuring out which ones mattered and which did not. Of course, all of these necessary changes had to be incorporated into our dump. Most of the other changes were ignored, replaced by better, more universal ways to do the same things.

Notable changes that had to be incorporated include handling of Pyramid's conditional symbolic links, both flavors of HP file system (14 character, and long file names), and the new, different kind of symbolic link in Mach.

## Testing

The backup system is one of the most critical pieces of software in any installation. Because of this, dump was tested extensively, on all of the different systems at each critical step in its development. Many file systems were dumped and copies restored, comparing the results. Tapes made with the old and new dump were compared for discrepancies. Finally, file systems with all sorts of unlikely structures were created and successfully dumped and restored, and these tapes compared with those produced by vanilla dump. The fact that the new dump is compatible with the old dump is not surprising, as it is based on the 4.3 code and idiosyncratic vendor changes were incorporated. In all of this testing, `restore` was never modified.

## Results

We have now been using this new version to do backups for about a year, with no problems whatsoever. Operator time has been cut by 50%; the operators are now spending more time walking tapes around then actually dumping information onto them. Reliability has also been increased; the only event that requires redoing an entire dump is a crash of the system being dumped (fortunately, a rare occurance).

## Availability

This version of dump will be available for anonymous FTP on tut.cis.ohio-state.edu as soon as most of the vendor specific code has been removed[2]

## Acknowledgments

Thanks to all of the operators at IICF, who constantly urged me to finish dump sooner. Thanks especially to Elizabeth Zwicky for proofing and encouragement, Steve Romig for encouragement, and Wayne Schmidt for proofing. I am most indebted to Diana Smetters for proofing, copious suggestions, and persuading me to actually *finish* this paper[3]

## References

[1] Benteley, Jon L., *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[2] Jacobson, Van & Karels, Michael, *TCP/IP Performance Improvement Tutorial,* Winter 1989 USENIX Tutorial M4, USENIX Assn., Berkeley CA, 1989.

[3] Joy, W., *et al., The UNIX Programmer's Manual,* Fourth edition, University of California at Berkeley, 1986.

[4] Leffler, S., McKusic, M., Karels, M., & Quarterman, J., *The Design and Implimentation of the 4.3BSD UNIX Operating System,* Addison Wesley, Reading, MA, 1989.

[5] Polk, J., & Kolstad, R., "A Faster UNIX Dump Program," *Proceedings of the Winter 1988 USENIX Confrence,* USENIX Assn., Berkeley CA, 1988.

[6] Rochkind, Marc J., *Advanced UNIX Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] Speck, Donald, Private communication (at Winter USENIX), 1989.

[8] Zwicky, Elizabeth, "Backup at Ohio State," *Proceedings of the 1988 USENIX Workshop on Large Installation Systems Administaration,* USENIX Assn., Berkeley CA, 1988.

[2]Thanks to Encore Inc. for allowing us to redistribute Sue LoVerso's changes for Mach fast symbolic links.

[3]She is still mad at me for not tuning dump back when she had to do the backups : –).

# YABS *

*Nick Simicich*
IBM T.J. Watson Research Center
Distributed Computing Environment Project
(914) 789-7033
njs@ibm.com, uunet!bywater!scifi!njs

June 29, 1989

## 1 Philosophy of YABS

### 1.1 Environment and Rationale

IBM T.J. Watson Research has a mature VM/370 environment [Doherty 86] which has been in place for years. While many IBM PC and AT class workstations had been installed, most of them had been relegated to use as terminal emulators. Many people felt that the mainframe environment was clearly superior, as response was consistently good, and central services were provided.

In order to get people to use workstations at Watson Research, we had to provide the services users were used to getting on the mainframe.

One critical service was backup. We had to make backup automatic for single user workstations, and this had to be done with limited programming resources. The IBM mainframes had storage resources that could be used to provide backing store[1].

The users running on the mainframes were accustomed to being provided with a list of backed up versions of files from which they could selectively restore. Similar function had to be provided for the workstation before the service could be considered an acceptable replacement.

### 1.2 Design Overview

This backup package is designed to back up a file system. It is not an image backup, and depends on the view of the file system as supplied by UNIX[2]. It will not back up unmounted devices, and will also not back up hidden files[3]. There is a certain amount of preparation which the user must do if the restore system is to be used to recover from catastrophe.

Some backup system designs require intelligence on the part of the storage system. They require support from the operating system they are storing into. YABS is designed to function on the user's workstation with a minimum of support from the target system using a commonly available data transport protocol which runs at high speed on the lan, FTP.

FTP could not completely isolate YABS from the operating system which ran on the CPU used for storage. Such things as file name space search, transport mechanism, mapping of backup names to the backing store's namespace, storage allocation and cleanup are not things with obvious, transportable answers.

*This can stand for Yet Another Backup System or Yorktown Automated Backup System. Take your choice.

[1]When I use the term "backing store", I refer to the system or systems that you are backing up to, and to the storage that that system stores the backed up data and catalogs into.

[2]Unix is a trademark of AT&T Bell Laboratories.

[3]AIX/RT has file over file mounts.

To provide flexibility, a modular design was adopted. Programs with external dependencies were put into separate units, with clearly defined interfaces. These basic building blocks can be replaced with user supplied programs which conform to the interfaces without losing the function supplied by the other building blocks.

The requirement that users be able to determine when they had versions of a particular file backed up was handled by keeping catalogs of backed up files. There are two types of catalogs, incremental and base. Incremental catalogs reflect a difference beteween that backup and the last backup, whereas base backups represent the beginning state of the system.

## 1.3 Backup Requirements

The user is responsible for installing the YABS package and for picking defaults. The user must also decide on a backup schedule. Once these decisions are made, the Unix cron [1] facility can be used to drive the backups, and notification of backup failure and success will be via mail communication with root. The user is responsible for checking these mail messages to determine if the backup has run successfully. Alternatively, the user can modify the incrback shell script to return a notification in some other way when the backup fails.

The package currently has some AIX dependencies. Because it uses uftp[4] to transport the file to backing store, it also depends on the availability of a TCP/IP communication system.

### 1.3.1 Backing Storage

YABS is designed to move the backed up data to another system using the FTP protocol which runs under TCP/IP. Thus, any system which is capable of storing a byte stream file and returning it unmolested (or at least not permanently damaged) can be used to store the data produced. Current candidates include VM, MVS, and other Unix systems. Since the user has control over how catalogs are mapped to backing storage locations, data can be migrated from one storage location to another, and the mapper function need only return the current location of that data for the restore operations to work properly.

### 1.3.2 Workstation Requirements

**Space Requirements.** A directory on the workstation must be set aside to contain backup programs, the backup control file, and backup catalogs. A base catalog on my system can be quite large, taking up to several megabytes of storage. This is dependent on the number of files which are being backed up. The default for this directory name is "/usr/backup".

Each incremental catalog is, when first created, as large as the base catalog. Once that incremental catalog is used as a base for the next incremental backup, it is stripped, and all file records which do not indicate that some change has taken place are removed from the catalog. Typically, for me, a stripped catalog is around 10,000 bytes. When a catalog is no longer required, it may be deleted. Generally, it is useless to keep catalogs which represent backups where the files that contain those backups no longer exist on backing store.

**Operational Requirements** The backup system's files and programs should not be accessable to any users other than the system administrator. Since the backup data is maintained in files owned by the system administrator, it should be presumed that the administrator can look at the data and can keep looking at the data for the life of the machine. Unfortunately, this means that the administrator will have to run all restores.

The (user supplied) mapper program may be required to supply a password to allow access to the file storage, and may be required to supply a separate password to allow the user access to the other system. The supplied command line is stored in the catalogs, and other information which may be of a confidential nature is also stored in the catalogs. Because of this, the catalogs should be protected from

[4] See page 5 for information about the uftp program.

access by other users. The backup stores the catalog with mode 600 (read/write only by owner) and this should not be changed without careful consideration.

Included in this package is a program called "asroot" which, with appropriate authority, will allow a user to run a command with root authority. Access to this program should be carefully controlled. The user may decide that in their circumstance, it is not appropriate to use the program at all, in which case the "incrback" shell script must be modified. A stanza to set this program up is included in the makefile, which will allow the program to execute with root permissions and be executable only by root or other people in system group. It is presumed that the system administrator, and only the system administrator, is in system group.

The usual problems inherent in LAN and TCP/IP aliasing apply.

**Time Requirements.** It currently takes me about four hours to do a complete backup of my system. My daily backups take around 20 minutes. The backup runs CPU bound, doing data compression. Data transfer rate is 12–17 kb/sec, depending on system load and compression. I actually back up around 400 meg of data, but that is an atypically large amount.

## 2 Building blocks

### 2.1 Backlist

Backlist is the program which is responsible for building the list of files that the user wants considered for back up on any particular day. It is designed to be a user replaceable piece. The defined interface is that it must produce, on standard output, the list of files to be backed up. The file names must be complete path names, from '/' (root), and must be separated by only a single null. This program should not fail. If it does, it should produce a non-zero return code, and, hopefully, an informative error message. The backlist program is passed a single parameter which is a flag, if specified by the user when backup is invoked. The flag is –i, and that flag, if present, is followed by a file name which is supposed to represent a control file.

#### 2.1.1 The supplied backlist.c

The supplied backlist.c fits the above description. In addition, it has the following characteristics:

1. It will not descend below a mount point where the mount point does not represent an actual device. This is true unless the mount point is mentioned in the path of an "only" or "also" control card.
2. The selection of files is controlled by the above mentioned control file. the format of the control file is explained below.

The control file consists of a series of lines. Each line may optionally begin with whitespace, and then a valid keyword must appear. the keywords are listed below. Following the keyword must be at least one space, and then a regular expression (which is considered to extend to the end of the line and not to include the newline). The regular expressions are in the form recognized by the regex() library call [2].

If one or more include statements are encountered, a filename must match one of the include statements to be considered for inclusion in the output list. If there is no include statement, then all files are considered for inclusion.

If any exclude statements are present, the filename must match none of those exclude statements to be placed in the output list.

The general form of a control statement is:

$$\left\{ \begin{array}{c} include \\ exclude \end{array} \right\} regularexpression$$

or

$$\left\{\begin{array}{c} \textit{omit} \\ \textit{only} \\ \textit{also} \\ \textit{einclude} \\ \textit{eexclude} \end{array}\right\} \textit{filenamepath}$$

Please note that regular expressions are not ordinary file matching expressions. Thus, the statement "include /u/*" matches "/u", "/u/", "/u//", "/u///...", etc., which is probably not quite what you want. The include statement to match all of the expressions in the /u directory is "include /u/.*". In general regular expressions, the "." means match any single character, and the "*" means match any number of them.

The

*filenamepath*

operand used by "einclude" and "eexclude" must match the file name to be included or excluded exactly. One might question the wisdom of having a new keyword for doing exact matches, when it is already possible to construct a regular expression that will match only a particular file.

The rationale for having more than one means of inclusion and exclusion is that when you specify file names exactly you can build a sorted table and search it with bsearch() [2]. For include/exclude lists of any size, this is significantly faster than searching a list of regular expressions, which must be searched sequentially.

The particular technique that this makes usable is as follows:

Say you have a problem with your /usr file system. You run fsck, and some small number of files are blasted. The output of fsck rolls off of the screen during boot, and as a result, you don't know which files have been blasted.

You issue a `find /usr -print >/tmp/listoffiles` command, and then you edit that file so that there is an eexclude in front of each file.

Then you add a line at the beginning of the list that says:

```
only /usr/
```

When you run that list through `restsel -s max -i/tmp/listoffiles`, it will rapidly produce a restore catalog that contains only those files that used to be there but are not now there. Before "eexclude", this operation typically took a couple of hours.

### 2.1.2 only and also

"Only" and "also" are used to select particular sections of the tree to back up. Only implies a filtering action, in that if any "only" control statement is specified, a file must match some "only" to be backed up or restored. "Also" implies addition, that is, in addition to the normal set of files that would be backed up. the normal set of files that would be backed up are the set of files that are on your machine and which are not mounted over. "Also" is designed to descend into files mounted from another machine, or locally mounted directory on directory.

As an example, I have a directory from another system called **/usr/watson** mounted on a system using a Distributed Services remote file mount. If I specify **also /usr/watson/**, backlist will descend into that file system and seect the files in that system as well. If that file system has further directories mounted into it, those directories will not be backed up unless they are also specified by "also" control statements.

Note that if "also" is used, and this causes files to be backed up twice, it will be possible to cause *yabs* to become confused regarding linked files. When specifying "only" and "also" control statements, be sure that each directory structure is only represented once in the backup tree.

## 2.2 pcterse

Pcterse is a file compression program, and is used for two reasons.

1. It reduces the size of the backup file.
2. It reduces the rate of the backup, and thus cuts down on network saturation.

The pcterse program was written at Yorktown Research, by Victor S. Miller for the IBM PC. He later ported the program to AIX.

## 2.3 uftp

The uftp program is used to transport the data from the backup system to the target system, using the FTP Protocol [RFC959] and can communicate with any FTP server that implements ports and binary mode.

Uftp implements an ftp client. All parameters required to perform one file transfer are implemented as options, and the data to be transferred can either come from standard input, or be piped into standard output. Uftp returns a zero return code if all aspects of the transfer complete successfully, and a non-zero return code if any aspect of the transfer fails. It does exactly one transfer per invocation.

## 2.4 mapper

The mapper program must be supplied by the user. The one supplied with the backup system is a sample only, although some effort has been put into making it handle many of the common cases. The mapper is called as a separate process and passed two arguments. These two arguments are the major and minor backup numbers. The major number changes every time that a base backup is done.

When the yabackup program runs, it transfers the backup as a number of separate files, where each file has a size limit of approximately ten megabytes before compression. That is, the backup system will cause a new file to be generated whenever there is another file to be backed up and the last file backed up pushed the current string over the ten megabyte limit.

For a base backup, the first request will be made with a base backup number of 1, and a minor number of 0. The information returned by that request will be used to back up the first ten megabytes of data plus the catalog. Then a new call will be made to the mapper, and the information returned by that call will be used to back up the next ten megabytes of data.

The first call for the base backup has a minor number of 0. Each incremental backup increases the minor number by 1000, so the backups are numbered

1,0

1,1000

etc. When the base backup makes a second call to the mapper it increases the second number by 1 so that the calls for the first base backup would be

1,0

1,1

1,2

...

whereas if the third incremental backup had a lot of data to back up, it would make calls with arguments

1,3000

```
1,3001

1,3002

...
```

The mapper program must return exactly two strings, separated by a newline character. The first string is the uftp parameter list used when the backed up data is stored, and the second string (following the newline) is the uftp parameter list that is used when the catalog is backed up. The second string will be used as an argument to printf() [2] and must have a %s in it so that the disk file name of the catalog to be backed up can be inserted into the returned string before it is passed to uftp.

These first set of these strings for any particular backup are saved in the catalog for informational purposes, but they are never used to retrieve the data. Instead, if the data is needed, the same two parameters will be passed to the mapper program, and the mapper should respond with the strings again. These parameters will then be passed to uftp (with a –i flag prepended instead of a –o) to cause the file to be read.

It is possible that the mapper will decide that a particular parameter set cannot be mapped. In that case, the mapper program should put out a reasonably informative error message on to standard error and exit with a non-zero return code. However, the mapper program should still do the mapping operation, and return the mapping strings on standard output if the parameters can be mapped at all.

An optional third argument of "restore" can be used to tell the supplied mapper.c that it is running a restore. In the current version of yabs, this argument is supplied when yaburest calls mapper. When "restore" is supplied as a third argument, all logic which might be called to delete any files on backing store is skipped, and thus mismatch between the yabs control files and what is to be restored will not cause files to be deleted unpredictably during a restore operation.

### 2.4.1 Cleaning Up.

After your backups have aged for a while, the data in them becomes less valuable. Eventually, it is time to figure out which ones are obsolete and to throw them away.

The cleaning up may well be the hardest part. First thing you have to do is to decide how many backups of what type you want to keep on backing store, and how many backups of what type you want to keep catalog pointers to.

The cleaning up seems to be intertwined with the mapping of backup numbers to backing store names. This is because:

- The mapping process must be aware of the data names that the backups are stored under.
- The mapping process must be aware as to whether a particular mapping can be done, that is, if any particular mapping is valid.
- The mapping process is specifically user controlled. Only the system administrator can decide exactly how many backups are needed and what they need to back up.

Thus, in the sample supplied mapper.c program, three strategies for chucking old backups and catalogs have been defined and implemented. They are explained below.

**Strategy 1: Don't worry about it.** You could let them accumulate forever on backing store. This might be the way you want to go if there is some automated scheme for removing them from the backing store after they get too old. There might be more overhead in deleting them from the backing store than is simply involved in allowing them to expire. In this case, the only thing you'd have to worry about is deleting the catalogs you have no active interest in, from your workstation.

You might also be copying your backups to a non-reusable media, such as WORM optical disk. In that case you can't delete them.

**Strategy 2: Backed Up Backing Store.** When mapper detects that the base backup number is changing, indicating that a new base backup is being started, it can delete all of the old catalogs and backups from the workstation and the backing store.

Since the backing store is backed up, your backup is backed up. You will have to manually restore the backup to where you can get it from backing store, once deleted, if you need to restore it to your workstation.

This strategy is implemented in the supplied mapper program by calling uftp to list those files that are on backing store. Further verification is done to insure that those file names really look like backups and catalogs, i.e., they are matched against regular expressions that should only exactly match those files. Then uftp is called to delete them.

Finally, all catalogs are purged from the workstation.

**Strategy 3: Periodic Reuse of Backing Store Files.** This scheme assumes that you have all of the storage you need to keep some fixed number of backups on your backing store. It also presumes that you want a more complicated mapping scheme.

What you do is decide how many backups you wish to keep on backing store, and how many backups you want to keep incremental backups for. Then you calculate the file names using a modulo scheme. Supposing that your backing store is on VM, and that you wish to keep four base backups with incrementals for the current base and the previous base.

This means that you can when you take your fifth base backup, you can use the same backing store file names as the first backing store used.

When you take incremental backups that are based on the third base backup, you can use the same file names that you used for the incremental backups that were based on the first incremental backup.

To use this scheme, mapper must be aware of what the current backup number is, of whether the backup is for a new file or for a file that needs to be retrieved, and of whether a request for retrieval can be honored. The mapper program can make these decisions by examining the backup.control file (the format of which is explained above) and deciding, based on that in comparison to the arguments, whether or not to return a zero return code after mapping.

The general rules followed by the algorithm are as follows:

- Insure that the parameters represent a recent enough backup.
- The second parameter determines if this is a base backup or an incremental backup. If 1000 or over, this is a call to map an incremental backup. If under 1000, this is a call to map a base backup.
- To map a base backup: $p_1 \leftarrow (p_1 \bmod n_1) + 1$ where $n_1$ is the number of base backups to keep.
- To map an incremental backup: $p_1 \leftarrow (p_1 \bmod n_2) + 1$ where $n_2$ is the number of base backups for which incremental backups are kept.
- If this is the first call for a new base backup, list and delete the old base backup catalogs, and the old incremental backup catalogs.
- Produce the uftp parameter strings.

### 2.4.2 /usr/backup/mapper.plist

is the name of the file that is used to control the supplied mapper program. It contains the following entries:

**system** The system to which you want to back up. This parameter will be passed to uftp using the –n parameter. Uftp will look up the system name in `$HOME/.netrc` and use the password and id found there.

**directory** The directory into which the backup files should be stored. This parameter is used by uftp to issue a cwd before storing the files. If you are backing up to VM, this should be specified as userid.mdiskaddr. Example: If I want to back up to my B91 minidisk on VM, I would enter this parameter as "njs.b91".

**password** If a password is required after changing to a directory (as by VM) then it should be specified here. If no password is required, specify "none".

**strategy** A numeric digit indicating strategy number. Either 1, 2, or 3.

**numbases** A numeric field indicating the number of base backups to be kept. This includes the current backup. Only applicable to strategy 3. Can be omitted for strategy 1 or 2.

**numincr** A numeric field which represents the number of base backups for which incremental backups are to be kept. This includes the current backup. Only applicable to strategy 3. Can be omitted for strategy 1 or 2.

## 2.5 yabackup

The yabackup program controls the backup process, invoking all of the other programs that are part of taking the backup. It can respond to several different flags:

**–i screening file** This file name defaults to /usr/backup/list, and contains the list of include and exclude statements which is used by backlist.

Set up the /usr/backup/list file. This file controls which files are to be backed up and which are not. My /usr/backup/list file (which is supplied as a sample) reads as follows:

```
exclude /usr/backup/.at[0-9]+.*
exclude .*/core
omit /usr/spool/mqueue/
eexclude /usr/adm/wtmp
exclude /usr/adm/ras/errfile.[0-9]
eexclude /usr/lib/cron/log
omit /usr/spool/news/
```

If you want to back up all files, you may skip this step. A warning message will be produced and all files in the system will be backed up.

**–b** The –b flag is specified to force a base backup. A base backup may be taken automatically by the backup system when an unusual event occurs, such as if it can't find the backup.control file.

**–l file list command** Causes this program to be invoked, instead of the usual /usr/backup/backlist to compile a list of all files to be included in the backup.

**–c catalog directory** Defaults to /usr/backup.

**–m mapper** Defaults to /usr/backup/mapper. Points to user specified mapper program.

**–v** Suppresses much of the output from the tar program.

The yabackup program operates in approximately this fashion:

1. A lock file is generated or examined and modified to insure that there is no other concurrently running backup process. This is done by all programs which actually manipulate and change the catalogs.
2. The backlist process is started.

3. The backup.control file is examined. Based on the contents of that file and the flags on the command line, the new major and minor numbers of the backup are calculated.

4. The mapper is called to return the transport information, and the catalog base record is written.

5. The output of the backlist process is passed, and is used to build the backup catalog for today. If it is a base backup, this is done directly, whereas if this is an incremental backup, the information derived is compared against the last incremental catalog.

6. For each ten megabytes to be backed up, the following is done:

   (a) The tar, pcterse, and uftp processes are forked.

   (b) The list of files to be backed up is written to a named pipe, which is read by tar.

   (c) The child processes are waited on and reaped.

   (d) If there is more data, mapper is invoked again to get a new mapping string, and so forth.

7. The catalog is backed up.

8. If all of the above works well, the backup is called good. the backup.control file is updated, and the lock file is removed.

## 2.6 catstrip

The catstrip program strips all incremental catalogs except for the one that is pointed to by the backup.control file. This is an optional step, and need only be done if storage is a problem. Normally, this program looks in /usr/backup for base backups, but it will accept the –c flag, allowing you to change the default directory to some other.

Normally, this should only be run when you are reasonably sure that the last backup completed successfully.

What does stripping a catalog actually do? Well, when one runs an incremental backup, the newest catalog contains an image of the state of your filesystem. This includes stat() [2] information for all of the files, where they have been backed up to, and so forth. If the information for any particular file didn't change, then this is just an image of the catalog entry for the prior day.

Why is all of this information kept? Honestly, it is kept because it makes the incremental backup simpler. The current file system state can easily be compared to note the differences. It also allows one to more easily restore to a particular state by restoring only one catalog, and then basing the restore on that catalog.

But this takes up a lot of space. On my workstation, it would take a megabyte per day to store the catalogs. Since much of this information is redundant and one does not need the prior files to perform the incremental backups, the catstrip program will remove redundant information.

Information is considered redundant when it does not represent a change, and there is a copy of it in a more current catalog.

It is possible to construct an image of your backed up files on any prior day by including all of the catalogs between the base backup and the incremental you are interested in. The only problem here is that this image will not properly track deleted files. In other words, if a file were created, backed up incrementally, and deleted, it would still appear in the merged composite image. You would have to look at the backup catalog and specifically exclude those files you did not want to restore.

## 2.7 catlist

The catlist program provides a detailed listing of a catalog. The catalog can either be presented as standard input or as a argument.

This program may also be used to list the catalog that results from the running of the restsel program, so that you can predict which files will be restored.

## 2.8 restsel

The restsel program selects catalogs from the available catalogs by either catalog number or recorded backup date. Then, using the same include and exclude syntax that is used by the backlist program (in fact, it is the same selection program) one can select particular files to be included in the result catalog.

The result of the restsel program is a combined catalog, called "cat.backsel", which is placed in the current directory. This catalog can be listed by the catlist program.

When one is satisfied with the catalog, it can be fed into the yaburest program, which will do the actual restore.

The syntax of the restsel command is:

**–s (operand)** Selects by catalog. Catalogs are selected by number, or by position. The operand can be:

**all** This is the default, and selects all catalogs.

**max** This selects the newest catalog only.

**"selection selection"** Allows the selection of catalogs by specific number. A selection can be a single number ($n$), a range of numbers ($n - n$), or a combination of those ($n - n, n, n - n$) separated by commas. A catalog has to match any of the first selection and any of the second selection to be selected. If the second number is not specified, all catalogs with that major number are matched. Examples:

–s '2 0' Base backup 2, only.
–s '1' All catalogs based on base backup 1, including the base.
–s '1–5,8 0' Catalogs for base backups 1 through 5 and 8.

**–t yy/mm/dd or yy/mm/dd–yy/mm/dd** selects backups that are recorded as being taken on these dates. If both catalog numbers and dates are specified, the union of the set of catalogs selected by each is used.

**–c catalog-directory** The directory in which the catalogs to be scanned are stored. The default is /usr/backup. The result catalog is always put into the current directory.

**–i** [– [selection-file-name]] If a selection file of include and exclude statements are to be used, this option must be specified. A dash "–" indicates that standard input will be read for include/exclude statements.

If a range of catalogs or dates is specified which does not include a base catalog, or an unstripped incremental catalog, only those files that changed during that time period will be represented in the combined result. Including a base catalog will insure that the file is represented in the result. You can also manually restore the backed up catalog from the date you are interested in (by using ftp or uftp directly). The catalogs are backed up before they are stripped, so they represent an image of the system, including all of the files in the base backup that have not changed since then, but not including any of the base backups.

If the restored catalog is restored to the catalog directory under the original name, it will be stripped the next time catstrip is run. This is likely to be desirable.

## 2.9 yaburest

The yaburest program actually does the restore operation. The "cat.backsel" file produced by the restsel program is read and the files named in it are restored. The following is the general method of operation used by the yaburest program:

1. The cat.restsel file is validated.

2. The cat.restsel file is read in and the catalog entry for each of the files represented in the catalog is copied into a memory block. This produces an ordered image of the restore catalog as a structure in memory. The structure is ordered by file name within backup number, in ascending sequence.

3. Each of the elements in the structure is compared to all of the subsequent elements in the structure, and if the file name and complete path is the same, the older element is deleted from the structure.

4. The remaining set of files is scanned. For those backup anchors with actual files still attached, the mapper is invoked to recalculate the uftp parameter string, and uftp, pcterse and tar are invoked to transport, decompress, and restore the data files.

5. After the regular files are restored, ownership, permissions, and special files are restored and/or created from the catalog information. Thus, the final files should match the original backed up files. Much of the resetting of the files (and the creation of special files other than pipes) depends on being superuser while the restore is operating.

The format of the yaburest command is as follows:

**–r directory** This is the directory that the restore is to be done into. It should be an absolute path name (beginning with a slash) and should describe a directory that already exists. It should have enough space in it to handle the restore. You can check on the amount of free space in a filesystem with the df [1] command. All backups are done as relative backups, (after tar is forked, it changes directories to root ("/")), so that, for example, /usr/backup is backed up as ./usr/backup. If this parameter is not specified, the restore will be done to /usr/backup. If this parameter is specified as, for example, /tmp, the restore of /usr/backup will be to /tmp/usr/backup.

**–l restore-catalog** This indicates the catalog that will be used to do the restore from. The default for this name is "cat.backsel".

**–m mapper-program** The mapper program is the program that is used to convert the major and minor backup numbers to uftp parameter strings.

## 2.10 backup.control

The backup.control file is built by the backup system. It is used by the backup system to track the last completed backup. The user is not required to create the backup.control file and should not modify it. It may be examined to determine the state of the backup. This is especially important for the mapper.c program. The backup.control file generally contains two numbers. These numbers are in ascii format, and delimited by "whitespace" so that they can be read using the C scanf() [2] library call or with the awk [1] command.

The first number in the file ($n_1$) is the number of the last successfully completed backup. The next number in the file ($n_2$) is the number of the last successfully completed incremental backup. The second number is zero in the case where the last successfully completed backup was a base backup.

The backup.control file is used as follows:

**No backup.control file.** When there is no backup.control file, the backup system presumes that it is starting for the first time. A backup.control file is created, and the magic numbers "99999999 99999999" are written into that file.

**A backup.control file that contains "99999999 99999999".** The backup system presumes that it restarting after a failed initial backup.

**$n_1 \neq 0$ and $n_2 = 0$.** The system presumes that the last successful backup was a base backup which is numbered $n_1$. An incremental backup will be done (unless a base backup is being requested) which will be numbered 1000.

**$n_1 \neq 0$ and $n_2 \neq 0$.** The backup system presumes that the last successful backup was numbered as given in the backup file. The next incremental backup number is calculated as follows:

$$new\ n_2 \leftarrow 1000 * \left( int\left(\frac{n_2}{1000}\right) + 1 \right)$$

The backup.control file is not updated until the backup completes, and the updated catalog has been closed and backed up. Thus, it is an indication of a successfully completed backup.

In the future, the backup.control file may be modified so that the state of a partially completed backup is tracked in the backup.control file. In that case, plans are that the backup.control file will be used in such a way that the first two numbers in the backup.control file will not change in meaning, and that additional information will be placed into the file after the current information.

## 3 Usage Examples

### 3.1 Sample Restore

**The case of the lost ".profile".** The hypothetical case is this: I never log off of my RT, I just physically secure it by locking the door to my office. Sometime in the last week or so, my .profile file was erased by the AIX file gremlins. I don't know when that was, but I'd like to restore it.

**su root** To establish authority to read the backup catalogs.

**restsel -i-** This defaults to reading all catalogs. It will then prompt you for an include/exclude statement, after reading the first catalog.

**include /u/njs/\.profile** Remember these are regular expression, not normal globbing expressions. Dot, unless escaped, matches any character.

**control-d** To let restsel know that you plan to enter no more input.

restsel will then run and create "cat.backsel". Since I wrote the program, I believe that it might have found more than one version of the .profile file, but that is the only file that it might have found.

**catlist cat.backsel** You can run this output through pg [1] to list the catalog and insure that it was selected correctly.

**yaburest** Reads cat.backsel, and eliminates all the duplicate files. This should leave it with one file, the most recent version of the .profile in question, which will be restored right into the /u/njs directory.

**The case of rm -r *:** In this case, I thought I was in directory /tmp/bin, but I was actually in /u/njs/bin. Since I'm careless, I entered the command anyway. About a tenth of a second later, I realized what I had done and hit the interrupt key, but the files had been deleted.

This is almost the same as the first scenario, except for two things:

- Instead of the most recent version, of a particular file, I want to restore the most recent state of this section of my filesystem.
- Any files I deleted before my last backup from this section of my filesystem, I want to stay deleted.

**restsel -smax -i-** This specifies that I want only the newest of the catalogs. It also specifies that I will be prompted for include/exclude statements.

**include /u/njs/bin/.***

**control-d** EOF.

restsel will then run and create "cat.backsel". It will represent the state of all matching files as of the last backup.

**catlist cat.backsel** To make sure that the selections were done correctly.

**yaburest** Reads cat.backsel, and determines which of the backup files each of the files to be restored should be pulled from. Runs a restore for each of the files needed. Restores the files directly into /u/njs/bin.

## References

[Doherty 86] Doherty, W. J.;Pope, W. G. *Computing as a tool for human augmentation*, IBM Systems Journal, Volume 25, Nos. 3/4, 1986

[1] *AIX Commands Reference Manual*, SC23-2011, SC23-2081, IBM Corporation

[2] *AIX Operating System Technical Reference, System Calls and Subroutines*, SC23-2125, IBM Corporation

[RFC959] Postel, J.B.; Reynolds, J.K. *File Transfer Protocol*. 1985 October

# An Optical Disk Backup/Restore System

*C. J. Yashinovitz*
*T. Kovacs*
*John Kalucki**

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

A system has been constructed to perform daily UNIX** file system backups to a Sony optical disk attached to a Sun workstation. Daily dumps are automatically started at night and written across the network. To help insure reliability a checksum is done on the data while the dump is in progress and this is compared to the checksum of the dump file on the optical disk. To aid in restoring single files, a database is constructed and updated each night, which relates a file name to a particular dumpfile. Because these dump files contain virtually every file that changes they also can be used as an archive.

* John Kalucki is currently a student at Carnegie Mellon University, Pittsburgh, Pennsylvania

** UNIX is a registered trade mark of AT&T

# An Optical Disk Backup/Restore System

*C. J. Yashinovitz*
*T. Kovacs*
*John Kalucki**

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

## *1. INTRODUCTION*

The rapid increase in disk storage space in the past few years has increased the time and effort necessary to perform file system backup tasks. These backups are often preformed during the day when resources should be allocated to user oriented tasks. The book keeping involved in keeping track of multiple dump files on a single tape as weli as multiple tapes for a given days dumps has proportionatly increased. The advent of large capacity WORM (Write Once Read Many) optical drives presented the opportunity to construct a backup system which can operate unattended at night when machines are less busy. The presentation of the optical media as a UNIX** file system yields a method for organizing the dump data in an orderly, self documenting way.

The idea of using WORM drives for archiving data is not new[1] but the system described in this paper is oriented toward conventional backup and restore capability. As an added benefit, a database is constructed using standard UNIX tools to quickly locate all dump files which contain a given file name. Hence this system has automatic archiving capability as well.

## *2. HARDWARE*

The optical disk drive is the Sony model WDD-3000 attached to a WDC-2000-10 controller. The high capacity and stability of the media, the experiences of others with this drive and the availability of commercial software, were several factors important in this choice. Two types of media are available: 3.2GByte constant linear velocity (WDM-3DLO) and 2.1GByte constant angular velocity (WDM-3DA0) disks. The constant angular velocity disks have a significant advantage in large stroke seek time (250msec vs 800ms, 1/3 full stroke[2] ) but our main concern is large capacity and we use the 3.2GByte disks.

The controller is connected to a Sun 3/160 via a Ciprico RIMFIRE 3500 SCSI Host Bus Adapter. The whole package, including the SunOS driver was purchased from Genesis Imaging Technologies, Inc., Valley Forge, PA.

* John Kalucki is currently a student at Carnegie Mellon University, Pittsburgh, Pennsylvania

** UNIX is a registered trade mark of AT&T

## 3. *SOFTWARE*

### *3.1 Overall Strategy*

The objective was to do unattended file system backups in the middle of the night. The nightmare of system administrators is to find that the backup files/tapes are not in order when a restore needs to be performed. To reconcile these two facts we realized early that we needed a check to insure that what the dump program wrote actually made it to the disk. The remote dump facility `rdump(8)` does not have this capability. We quickly determined that NFS was too fragile in our environment for this robust task. We also had no desire to modify the dump and/or rdump programs and wanted an implementation that used commonly available UNIX tools as much as possible. Our solution uses shell programs to provide the overall control, rsh to transport the information across the network and two small C programs to perform checksums.

### *3.2 Sony WORM Driver*

The SunOS driver[3] makes each side of a disk into a 1.6GByte UNIX file system. Files can be created and manipulated in the same way as on conventional media but because the optical media cannot be erased, removing files consumes more disk space rather than frees it. There is 4KB of overhead in the creation of an optical disk file so it is not suitable for large numbers of small files. In our case this overhead is negligible because dump files are typically 100's of kilobytes to many megabytes long.

### *3.3 Wdump and Checksum*

A shell script called 'wdump' (worm dump) and two companion 'C' programs called 'wdumpremotesum' and 'wdumplocalsum' form a front end to `dump(8)` for moving data across the network in a very reliable way.

Wdump generates a unique dump file name from the local host name, the date and the dumplevel. Next it calls `dump(8)` and pipes it's output through wdumplocalsum and finally to `cat(1)` via `rsh(1)` for transmission of the data to the optical disk server:

```
/etc/dump "$LEVEL"udf 10000 - $FSYS | \
/etc/bin/wdumplocalsum $LOGFILE "Start $LEVEL $FSYS $SYSTEM '/bin/date'"|\
/usr/ucb/rsh "$DUMPSYSTEM" /bin/cat "> $DUMPFILE"

#  '-' tells /etc/dump to send its output to stdout.
# $DUMPFILE contains the full path of the file on the optical disk to
# dump to.  For example: /od00/physics/usr1b/May.5.1989.7
# $DUMPSYSTEM is the name of the optical disk server.
# Note the inclusion of a comment in the command line for wdumplocalsum.
```

When `dump(8)` exits, wdumplocalsum writes its checksum and size results, plus its command line comment argument to the logfile. Then it uses `rsh(1)` to run wdumpremotesum on the dump output file which has accumulated on the optical disk. The result of the remote checksum is appended to the logfile:

```
$ tail -4 /usr/adm/wormdump
39865 440320 Start 7 /      physics Fri May  5 02:08:49 EDT 1989
39865 440320 End   7 /      physics Fri May  5 02:09:41 EDT 1989
44711 409600 Start 7 /usr1b physics Fri May  5 02:11:57 EDT 1989
```

```
44711 409600 End   7 /usrlb physics Fri May  5 02:14:20 EDT 1989
```

Finally, If the checksums are different wdump exits with 1, otherwise 0.

Wdump is called with two arguments: the level of the dump and the filesystem to be dumped.

Wdumpremotesum and wdumplocalsum are modifications of the standard `sum(1)` command. In the case of wdumplocalsum, sum.c was changed to read from the standard input, and to copy standard input to standard output. Two extra arguments were also added: the output file into which the checksum is to be appended, and an optional comment to include on the same line with the checksum in the output file.

Both wdumpremotesum and wdumplocalsum report the amount of data which has been checked in bytes rather then in disk blocks, thus providing a greater degree of resolution. This is the only difference between wdumpremotesum.c and sum.c.

*3.4 Daily Dump Script*

In order to complete the automation of the backup process, a ksh[4] script called 'daily.dump' is run early every weekday morning by `cron(8)`. This script runs wdump described above and has the following features:

1. Uses `ping(8)` to make sure the optical disk server is up before beginning.
2. Reads /etc/fstab to determine what filesystems to dump. If the freq[5] field is 50 or greater, that entry is not dumped.
3. Determines the dump level automatically from the day of week.

   If the day of week accidentally turns out to be Saturday or Sunday, daily.dump complains and dies. This is because we do level one dumps manually to tape on the weekend. But the script can be easily modified to do a lower level dump on Saturday or Sunday if necessary
4. Checks /etc/dumpdates for consistency before and after wdump is called.

   On all weekdays except Monday, before the actual dump is run, the values of yesterday's day of week and yesterday's dump level, along with the device name of the disk the filesystem is on, are searched for in /etc/dumpdates. If they are not found a warning is sent to the administrators.

   The same sort of proceedure is used after each dump, this time with today's day of week, dump level and device, to be doubly certain the dump ran correctly. If not, a warning message is sent to the administrators.
5. Estimates the size of dump before each is actually run, to determine if it will fit on the optical disk
6. If daily.dump is called with a filename argument, backup commences with that filesystem and continues with those that follow in /etc/fstab. This is useful if either the optical disk server or the system being backed up crashes during the time daily.dump is running.
7. Checks that /etc/dumpdates has some rational size.

This was added after we lost /etc/dumpdates in a disk crash, and the next days dump filled a good portion of the optical disk.

8. Sends mail to root with all errors and warnings and waits on errors that are considered to be correctable. After the error condition is corrected daily.dump can be restarted with signal 3 (see Errorwait below).

Three levels of errors/warnings are supported: Warning, Errorwait, and Error. Each is invoked with a `ksh(1)` function of the same name.

Warnings are simply that. Some condition which we want the administrators to be aware of, but for which we do not want to interrupt the dumps. The 'Warning' function therefore just sends mail to root.

Errorwait is called for most error conditions. It sends mail to root, then goes into a sleep cycle. It sleeps for 10 seconds at a time, and during each awakening, checks to see if shell variable 'CONTINUE' equals 'true'. CONTINUE is only set to 'true' in a trap on signal 25. The lock file, whose path is stored in shell variable 'LOCK', serves a double purpose: it prevents two instances of daily.dump from running, and it contains the 'kill -25' command with the appropriate process ID number to restart daily.dump. The following is the relevant code from function Errorwait:

```
function Errorwait
{

/usr/ucb/mail root <<!          #mail args to root with generic message
$*

daily.dump is sleeping. To  restart, fix the problem, then execute $LOCK
and the script will continue where it left off.
!

trap "CONTINUE=true"  25         #if we get sig 25 set CONTINUE...
CONTINUE=false                  #       otherwise CONTINUE is false
COUNTER=0                       #COUNTER will tell us how long we are sleeping

until [ "$CONTINUE" = "true" ]                 # < - - - - - - - - #
do                                                               # W
let "COUNTER=COUNTER+1"                                          # a
if [ $COUNTER -eq 5400 ] #5400 * 10 = 54000 seconds = 15 hours   # i
then    # Notice Error is used to bail out completely            # t
        Error Nobody took care of dump script in 10 hours...     #
        /bin/rm -f $LOCK; exit                                   # L
fi                                                               # o
                                                                 # o
/usr/bin/sleep 10                                                # P
done                                           # < - - - - -  - - #

# If we ever leave this loop, it means sombody restarted us.
# function 'Warning' is a convenient means to communicate this.

Warning Ok... dump script continuing.
}
```

Finally the function Error is used if the condition is bad enough to halt dumping altogether. It is called if nobody services an Errorwait condition in 15 hours, and for certain invocation problems, i.e. somebody ran daily.dump on Saturday or Sunday.

*3.4.1 Restore.gen* In order minimize stress during a full filesystem restore, yet another `ksh(1)` script called restore.gen is provided. It generates a very simple script, which is easily checked for correctness, to issue the appropriate `rrestore(8)` commands to restore a filesystem as needed on that particular day. Normally restore.gen is called with a single argument, a file system name. If the -s (show) option is given before the file system name, the revelent dumps are displayed, and no script is generated.

```
$ restore.gen /usr3a
Output written to restore.script.
$ cat restore.script
#!/bin/ksh
#
# restore.script   Generated by restore.gen   Tue May 9 11:12:48 EDT 1989
#
# The following are the relevant dumps as determined from
# /etc/dumpdates:
#
# /dev/rxd41h       0 Sat Apr  8 15:34:59 1989
# /dev/rxd41h       1 Fri May  5 17:15:17 1989
# /dev/rxd41h       3 Mon May  8 01:36:29 1989
# /dev/rxd41h       4 Tue May  9 01:34:35 1989
#
# Anything lower then Level 2 must be dealt with by hand.
# Presumably these are on tape.
# If the above list looks good, and the tape restores are
# done, you are ready to proceed.
#
# This script expects to be run from the directory being restored!
#

if [ $PWD = "/" ]
then
        echo This should not be run from /, exiting.
        exit
fi

echo make sure photon:/od00/physics/usr3a/May.8.1989.3 is ready, hit return
read cr
echo /etc/rrestore rf photon:/od00/physics/usr3a/May.8.1989.3
/etc/rrestore rf photon:/od00/physics/usr3a/May.8.1989.3

echo make sure photon:/od00/physics/usr3a/May.9.1989.4 is ready, hit return
read cr
echo /etc/rrestore rf photon:/od00/physics/usr3a/May.9.1989.4
/etc/rrestore rf photon:/od00/physics/usr3a/May.9.1989.4

echo restore.script finished.
```

### *3.5 Archive Data Base*

This database was not one of the original objectives of this project. However it isn't long before hundreds of dump files have accumulated and the inevitable request comes to restore a lost file. But where is it? There was a clear need for an index into this mass of data. The idea was to construct a database which would indicate which dumps contained files of a given name. Under the principle of using only "standard" UNIX tools, this was accomplished by using the file system as a indexing tool and a few awk and shell programs.

The database is a directory which contains the subdirectories a,b,c,...z,0,1,2...9 and "other". Each of these directories in turn contains the files a,b,c,...z,0,1,2...9 and "other". Each morning (via cron) a script is run which sorts the output of a restore -t on the new dump files into the database. The key into the database is the first two characters of the file name. An entry for a file named "abacus" would be made in the database file a/b. A file named "november" would result in an entry into n/o. Non alphanumeric keys are mapped to "other" and upper case is mapped to lower case by file system links. The file a/A is linked to a/a for example.

A database entry consists of three tab separated fields. The first entry is the name of the dump file, the second is the name of the file that was dumped and the third is the path prefix to the file that was dumped. To save space a field which is identical to the same field in the previous entry is chrushed to a single asterisk. As an example consider these lines of output from "restore -tf /od00/physics/usr/May.3.1989.5":

```
36871       ./lib/uucp
19141       ./lib/uucp/uucleanup
19122       ./lib/uucp/uucico
18799       ./lib/uucp/uuxqt
19133       ./lib/uucp/uusched
19137       ./lib/uucp/uucheck
```

These lines generated the entries marked in bold in the database file u/u:

```
/od00/physics/usr/May.3.1989.5  uucheck ./lib/uucp
*       uucheck.o       ./src/lbin/uucp
*       uucico  ./lib/uucp
*       uucleanup       *
*       *       ./spool/uucp/.Admin
*       *       ./spool/uucp/.Old
*       uucleanup.o     ./src/lbin/uucp
*       uucp    ./bin
*       *       ./spool/mail
*       uucp.o  ./src/lbin/uucp
*       uucpdefs.o      *
*       uucpname.o      *
*       uuname  ./bin
*       uuname.o        ./src/lbin/uucp
*       uusched ./lib/uucp
*       uusched.o       ./src/lbin/uucp
*       uustat  ./bin
*       uustat.o        ./src/lbin/uucp
*       uux     ./bin
*       uux.o   ./src/lbin/uucp
*       uuxqt   ./lib/uucp
*       uuxqt.o ./src/lbin/uucp
```

New entries are inserted at the top of the database files so that subsequent searches will produce output in time reverse order, last in - first out.

The awk program dfind (dump find) is used to search the database. It requires one argument, the name of a file, and returns a list of dump file names which contain a file which matches it's argument. "dfind uucico" produces:

```
/od00/physics/usr/May.3.1989.5          ./lib/uucp/uucico
/od00/physics/usr/Mar.30.1989.6         ./lib/uucp/uucico
/od00/physics/usr5a/Mar.30.1989.6       ./1156/dunne/uucico
/od00/physics/usr5a/Mar.30.1989.6       ./others/htan/uucico
/od00/physics/usr/Jan.27.1989.7         ./man/cat8/uucico.8
```

The output from dfind can be further restricted by filtering with grep. Our current index uses 8.6MB and covers ~4.8GB of dump files.

## *4. Implementation Notes and Problems*

The overall dump processes are not controlled by the optical disk server. Each machine runs it's own dump script at a predetermined time. Because of the very long seek times on the optical disk it is clearly not efficient to have more than one optical file being written at the same time, so some care must be used to coordinate the dumps of multiple machines. Another level of control could be added to let the optical disk server run the dumps on remote machines, or some kind of blocking mechanism could be implemented which would allow only one active dump, but we have not seen the need and have not done so.

We still do epoch and level one dumps to tape. This is partly a development precaution and partly economics. We anticipate putting the one's on optical disk in the near future but the epoch dumps will probably have to wait for higher density disks. A 300MB dump and restore has been successfully performed so there does not seem to be any fundamental problems with large dumps.

The omission of a level two dump in our normal series is not an oversight. If things should go astray a level two dump can be performed to bring things back into line. The level two will go back to the last epoch or level one. The next day the normal daily dump will go back to the two.

Except for the gap between the tape and optical dumps the optical data is a complete archive of all changes to all files in 24 hour increments. We are considering ways to close this gap which would make the data set complete.

The optical disk software is not totally bug free. We have had several crashes of the Sun optical disk server in the past four months of continuous service that could be attributed to the optical disk driver. To date only one crash has caused a significant loss of data. A large file was being written to the optical disk via NFS at the time. Many restores have been run on NFS mounted dump files without incident so it seems that only writting poses some risk. These problems have been referred to the supplier.

## *5. Conclusions*

We have constructed a file backup system using commercially available components and standard UNIX tools. We have not altered the time tested dump and restore programs. In fact the programs dump and restore could be replaced relatively simply (with cpio on System V for example) without any loss of functionality. Currently this system is being used to back up 24 file systems with a total capacity of ~5GB on two machines and runs unattended, at night.

## *REFERENCES*

1. Andrew Hume "The File Motel - An Incremental Backup System for UNIX," *Proceedings of the Summer 1988 USENIX Conference.*
2. Sony sales literature.
3. The Optical File System software was written by Brian Swafford, Q Systems
4. Morris I. Bolsky and David G. Korn, "The Kornshell Command And Programming Language"
5. See 4.x BSD manual page for /etc/fstab (section 5)

## The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

*Computing Systems*, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Telephone: 415 528-8649
Email: office@usenix.org